

2(mix)
NASA CR-122469

**COMPUTER ENHANCEMENT
THROUGH
INTERPRETIVE TECHNIQUES**

**PREPARED FOR
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND 20771**

**FINAL REPORT
NASA GRANT NGR 33-022-125**

JANUARY, 1972

**Garth Foster, Principal Investigator
with
Henk A. E. Spaanenburg
Werner E. Stumpf**

(NASA-CR-122469) COMPUTER ENHANCEMENT
THROUGH INTERPRETIVE TECHNIQUES Final
Report G. Foster, et al (Syracuse Univ.)
Jan. 1972 89 p

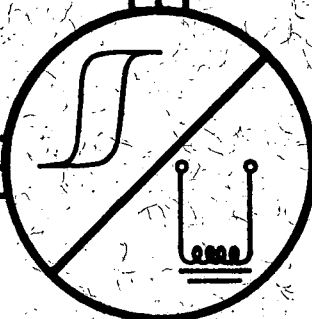
CSSL 09B

N72-31229

Unclas
G3/08 40466

Reproduced by
**NATIONAL TECHNICAL
INFORMATION SERVICE**
U S Department of Commerce
Springfield VA 22151

**ELECTRICAL AND COMPUTER ENGINEERING
SYRACUSE UNIVERSITY
SYRACUSE, N.Y.**



Computer Enhancement Through Interpretive Techniques

Final Report

NASA Grant NGR 33 - 022 - 125

January 1972

Garth H. Foster, Principal Investigator

with

Henk A. E. Spaanenburg

Werner E. Stumpf

Department of Electrical and Computer Engineering

Syracuse University

Syracuse, New York 13210

TABLE OF CONTENTS

1.0	INTRODUCTION	1
2.0	THE COMPUTER ENVIRONMENT	3
3.0	SCOPE OF THE PROBLEM	5
4.0	PRIMITIVE CONSTRUCTS	7
4.1	Timing Considerations	11
4.2	Space Requirements	12
4.3	Scalar Functions Extended to Vectors	15
4.4	Scalar Functions Extended to Matrices	17
4.5	Summation	19
5.0	MATRIX INVERSION AND LEAST SQUARES TECHNIQUES	20
5.1	Results	21
5.2	Summary	25
6.0	CLOSED PARTITIONS ON THE STATES OF FINITE STATE MACHINES	27
6.1	Translating from FORTRAN to <i>APL</i>	29
6.2	Results for Time and Space	41
7.0	THE FAST FOURIER TRANSFORM	42
7.1	Tests and Results for the FFT	48
8.0	A NASA APPLICATION PROGRAM	50
8.1	Program Characteristics and Programming Problems	51
8.2	Recasting the Original <i>APL</i> Program	52
8.3	Size of Computations and Their Implications	59
9.0	CONCLUSIONS	
	REFERENCES	65
	APPENDIX A	
	FAST FOURIER TRANSFORM PROGRAMS <i>APL</i> and FORTRAN	67
	APPENDIX B	
	THE FORTRAN VERSION OF BEAM FOR THE NASA RADIATION PATTERN PROGRAM	74

FIGURES

6.1	SP Functions	28
6.2	FORTTRAN Flowchart	30
6.3	Translation Steps	33
6.4	Subroutine REDUCE	39
6.5	Subroutine SUM	39
6.6	Subroutine NORIZ	40
6.7	Subroutine EQUAL	40
6.8	Subroutine LESS	40
6.9	Moments for REDUCE	43
6.10	Moments for SUM	44
6.11	Moments for NORIZ and EQUAL	45
6.12	Moments for LESS	46
8.1	<i>BEAM</i> (ORIGINAL)	53
8.2	<i>BEAM</i> (MODIFIED)	57

TABLES

4.1	Primitive Constructs Timings	9
4.2	Primitive Constructs Space Requirements	13
4.3	Linear Fit for Vector ADD	16
4.4	Quadratic for Matrix ADD	18

COMPUTER ENHANCEMENT THROUGH INTERPRETIVE TECHNIQUES

1.0

INTRODUCTION

This study had as its thesis the improvement in the usage of the digital computer through the use of the technique of interpretation rather than the compilation of higher ordered languages. Consequently, we have concerned ourselves on the one hand with the efficiency of coding and execution of programs written in higher ordered languages such as FORTRAN, ALGOL, PL/I and COBOL. Programs written in these languages are compiled or translated to the machine language of a specific machine and run in a production environment, generally that of multiprogramming.

For this study, we have selected FORTRAN as the high level language in examining programs which are compiled. Widespread use of the language, particularly for problems of a scientific nature, and the extensive numbers of implementations of the language over many years, clearly make FORTRAN a logical choice. While considerable experience has been gained in working with and creating compiler implementations for higher level languages, success reduced interest in the design of languages for which reasonably efficient execution in an interpretive implementation might be expected.

It would be useful if a study could have been made dealing only with general parameters of languages which effect either compilation or interpretation. It was felt that this was not possible, and a terse, powerful language was needed as the choice for the interpretive portion of this study.

For the interpretive language we chose A Programming Language, or Iverson's notation as it has sometimes been termed. [1,2,3,4]

The reasons for this choice are: 1) The language is rich in function, allowing for a compact notation for defining programs and intuitively offering a high compression ratio between source and a compiled equivalent. 2) In the *APL* interpreter the defined functions (programs) are stored nearly in source code, while the data and constants are stored in an internal format giving maximum compactness for both program and data. 3) The *APL* Terminal System is oriented towards processing regular arrays of data offering the possibility of minimizing interpretation overhead. 4) The primitive functions have been optimized due to hand coding in the assembler **language**.

The rationale of this study was that there are three areas where interpretive techniques could enhance the performance of computers. The first would be in those instances where interpreters could best compilers in execution speeds. Investigating such a possibility implies the restriction of the problems to areas in which both techniques could be applied and of course the use of higher level languages in coding the problems.

The second way in which utility could be provided by interpreters is that of trading machine cycles or execution speed for space in the run time code stream. The third way in which interpretation techniques would be of value would obtain if the implementation of an interpreter of a given language provides more effective use of programmer time in the development of software and for problems which are to be run once or only a very few number of times. In this context it is envisaged that a given language would have two (and perhaps more) implementations; one would be an interpreter on which the program development would be done and the other would be a compiler in which the production work would be done. If the problem is to be run few enough times, then the interpreter only would be used. Here the number referred to as a few depends upon the size and complexity of a program, the execution and compile time in addition to the interpreted run time; the cost of the program development, and the number of compilations used before the program may be run usefully for the first time. The three points

of view relative to interpretation given above sketch a range of capabilities ranging from direct superiority to sometimes usefulness.

In this report a knowledge of *APL* and FORTRAN is assumed.

2.0

THE COMPUTER ENVIRONMENT

The equipment and machine configuration on which this study has been conducted is Syracuse University's IBM System/360 Model 50 I (512 K bytes) with 2 2314 disk units. The operating system is the Syracuse University Operating System (SUOS), a modification of multiprogramming with a fixed number of tasks (MFT II) release 18.6, of OS/360 using a HASP-like spooling program to provide spooling and allocation of ports to interactive problem processors. Currently, SUOS is at the level of Release 7, modification 2. All computer runs were made between September 16, 1970 and September 15, 1971, and this period covers the time frame when *APL* was available as Program Product in its initial form, (XMI), and as a later, enhanced version, (XM6), both operating under Operating System /360 (OS/360). The FORTRAN H system is also available as a current IBM Program Product. Optimization was set to OPT=2, or the greatest level, for all FORTRAN runs except for the case dealing with the partitioning of finite state sequential machines. This case will be detailed later.

Although the FORTRAN programs were developed, debugged, and timed in a multiprogramming environment, times reported were measured in a pure rather than a batch environment. The same practice was followed for the programs developed in *APL* by use of the *APL* Terminal System. Thus, in the pure environment *APL* is up, when *APL* is being measured and there are no other *APL* users on the system, nor are there any batch users on the system. When FORTRAN is being measured in this environment *APL* is not up and no other batch users are on the system. Ranges of measured times between the two

modes are comparable, but measuring times in a pure environment

- 1) Gives repeatability to within the resolution of the timer and reduces the necessity of running many tests to obtain statistically measured times.
- 2) The problem of interference from and with other programs is minimized reducing, for example, the swap time attributable to them.
- 3) Minimizing the confluence in an absolute sense, as done here, produces an approximation of a batch APL which may then be compared to normal batch mode processing in a higher ordered language.

All measurements were made using the software monitors provided by the system. Since these were based on the system timer for the Model 50 which has a resolution interval of 16.67 milliseconds (1/60 of a second), some variations in times, even in the pure environment, will be encountered when the absolute times are small. These deviations are due, in part, to the software overhead in recording the times in addition to the problem of resolution. In general, the times measured for the two modes were sufficiently different and of a size that the error in making measurements in this manner was either not severe, or was reduced by measuring larger samples.

Program sizes in both modes of investigation are covered later but system sizes should be noted, FORTRAN H required partitions of about 160 K bytes. APL, in this system, requires 178 K bytes, if two workspaces are kept in core at a time (the minimum possible) and 216 K bytes if 3 workspaces are kept in core. The size of the workspace in both cases is 36 K bytes, a size which has become a defacto "standard" for APL\360. Some variations from IBM estimates of core requirements are to be noted for this system because SUOS allocates physical ports to APL and additional space is required for the interface. The nominal size requirements [5] are given by the estimates:

$$SIZE \leftarrow 88000 + (336 \times PORTS) + INCORE \times 8 + 2048 \times (WSSIZE \div 2048)$$

That is to say the amount of core in bytes required is 88000 for the interpreter and supervisor plus the storage required for terminal


handling (336 bytes per port) plus the number of workspaces in core times two words (8 bytes) more than the size of a workspace rounded up to the nearest 2 K boundary. The 36000 bytes choice for *WSSIZE* provide about 32000 bytes to the user.

3.0 SCOPE OF THE PROBLEMS

In any study there is always the question as to whether the range and the choice of problems are meaningful. We have chosen five areas for consideration and these are: 1) Primitive constructs, 2) Matrix inversion and operations on systems of linear equations, 3) The partitioning of the states of a finite state sequential machine, 4) The Fast Fourier Transform (FFT), and 5) A program for calculating the radiation pattern of an antenna with parabolic geometry. The last case was a program developed at Goddard by a visiting scientist and represents a typical application area at the Goddard Space Flight Center.

Examination of primitive constructs seeks a rough measure of relative efficiencies between *APL*, as an interpreter, and FORTRAN producing a compiled code stream, for simple computational constructs. The purpose of comparing primitive expressions was not an attempt to produce an absolute measure of power. Indeed, the constructs which were chosen are so simple that they are not likely to be individually significant in real life. They attempt to give insight into interpretation versus compilation in places where concise *APL* expressions, primarily reductions, dealing with vectors or matrices substitute for one or more DO loop structures in an equivalent FORTRAN program. The next point of examination was to consider the trade-off found in the interpreted environment (*APL*) between using a primitive construct such as scalar dyadic functions extended to arrays versus performing the function in a FORTRAN-like manner, with loops and operating on

scalars, while using an interpreter.

The second type of problem, matrix inversion and least squares techniques, gives a fairly complex situation, the programming for which has become more and more standardized. Matrix inverse routines are found in most scientific subroutine packages for the compiled environment and their use in that mode makes the library an important point of study when considering interpreters (essentially a library of routines) versus compiled code. Here DOMINO () was compared with matrix inverse routines found in the Scientific Subroutine Package as well as with Gauss-Jourdan and Gauss -Siedel routines written in *APL* and in FORTRAN.

The third area, finding all partitions of a finite state sequential machine having the substitution property, is one that is matrix oriented in formulation but iterative in solution. The problem can be handled through batch programming techniques but an interactive approach is most useful. The problem had been programmed elsewhere in FORTRAN on the Michigan Terminal System and then programmed by one of the authors (GHF) in (*APL*). Both implementations were turned over to another author of this report (H.A.E.S.) who at the time knew the algorithm for solution and was proficient in ALGOL but who had only then begun to learn FORTRAN and *APL* . The goal was to obtain measures of efficiency of coding in time and space and to test the readability of code in both systems. Additionally, the ability of translating from FORTRAN to *APL* is commented upon. For the examples chosen the space requirements are not pressing in either system. The *APL* written versions attempt to make the best use of the array feature of the language although there may be some limitations because of the problem.

The Fast Fourier Transform, in Case 4, is another situation where array capability plays a role and yet where an iterative process must be applied. Here a version of the FFT published

originally in ALGOL was translated to FORTRAN (by WES who knew FORTRAN and *APL* but not ALGOL) while the *APL* version was an improved version of a previously published FFT written in *APL*. In this case as with the previous one, some degree of program writing or translation may be inferred along with the results quoted for space and time requirements. In this case the space requirements for data storage in *APL* hamper the size of the FFT which may be used in that environment. While we examine the results obtained both in *APL* and in FORTRAN under the restriction that the data must fit in a 36K byte workspace (about 32K bytes available to the user), no projection is made to larger data sizes. Primary interest in the programming task was programming ease, program size and relative efficiency.

The final task an antenna field problem, as mentioned previously, was originally programmed in *APL* as a development model for the running version of the program which was coded in FORTRAN. In the present context the original *APL* function, and the report which was written to document the work performed by the NASA researcher, were used to rewrite the program to take advantage of the array capabilities of *APL*. The size of the space needed for data far exceeds the capabilities of storage in a normal system when attempting to make full use of the array orientation of *APL*. An approximation of speeds is made on the basis of smaller programs however.

4.0 PRIMITIVE CONSTRUCTS

The initial results in examining some of the primitive constructs are summarized in Tables 4.1 and 4.2. Ten examples are considered and a cursory examination shows that a number of cases deal with plus and times reduction. The reduction operator applied to vectors is equivalent to a single DO loop in FORTRAN and the times and plus function have often been quoted as measures of "computer power" so that add and multiply times for

popular computer systems are generally well known. Both functions have common counterparts in mathematical notation namely the summation over (Σ) and product (π) notations.

All cases are easy to understand and enter into the *APL* Terminal System. The same expressions when coded in a FORTRAN main program did not require an excessive amount of coding time but in at least one case each there was some choice (Case 8) and some difficulty (Case 10) in coding the subscripting in the DO loops.

Before direct comment is made on the times and space requirements, it should be noted that in addition to taking added time to code, the FORTRAN debugging times were longer due to what generally amounted to nearly a 24 hour turn around on program runs. This was because FORTRAN H, OPT = 2 was being used and the required region size of 160 K was not available continuously throughout the day. Consequently, no time spans to code and debug the equivalent FORTRAN programs for these ten expressions are given.

A longer time to code and debug the equivalent FORTRAN expression program was found for other tasks, as well as for this one, but no comparisons are offered due to the small number of programmers involved and the variations in programming skill and experience among those persons involved in this study.

TABLE 4.1
Primitive Constructs
TIMES in 60's of a second.

	APL		FORTRAN	
	60's of a second		CLG *	GO
1) +/11	1.45	I*4	580	16
		R*4	604	17
		R*8	615	19
2) +/12000	46.7	I*4	588	17
		R*4	623	24
		R*8	605	24
3) +/2000p1	44.1	I*4	615	16
		R*4	574	12
		R*8	583	17
4) +/17500	165.6	I*4	600	16
		R*4	614	48
		R*8	644	48
5) x/11	1.76	I*4	574	18
		R*4	590	18
		R*8	595	19
6) x/156	3.9	I*4	611	15
		R*4	608	22
		R*8	591	17
7) x/2000p1	50.7	I*4	596	23
		R*4	604	26
		R*8	599	23

* Compile Load and Go

		APL		FORTRAN	
		Time in		CLG*	GO
		60's of a second			
8)	$+/(11000) \in 11000$	4086.6	I*4	997	397
			R*4	973	390
			R*8	991	389
			$1 \leq J \leq I$		
			I*4	1070	477
			R*4	1052	478
			R*8	1066	476
			$1 \leq J \leq 1000$		
			I*4	700	17
			R*8	692	16
10)	$D+.LD + 5.5 \cdot 5pi \cdot 125$	74	I*4	1086	41
			R*4	1022	36
			R*8	1095	39

*Compile Load and Go

Table 4.1 Continued

4.1 Timing Considerations

Case 1 and Case 5 represent the overhead in each case in setting up the looping mechanisms and terminating the processes in question. Cases 2 and 3 represent a moderate number of components (in terms of the size of the workspace). In Case 3 the data is easier to generate but packing and unpacking the data takes place between generation and reduction. Number 4 approaches the upper size of vector of intergers which may be generated in a 36K workspace. The sixth expression is limited by the largest factorial which may be exactly calculated using long precision arithmetic. The seventh expression may be compared to number 3 in terms of changing the function of reduction. Cases 8, 9, 10 represent more complex problems in data generation, searching, and inner product. There is no significance to the choice of the functions used in the inner product except that minimum was chosen as a reasonably simple primitive requiring either some additional coding in FORTRAN or a call to a FORTRAN library routine.

For reductions over vectors with a small number of components, *APL* is faster than the execute step of the compiled FORTRAN. In these cases the careful hand coding required of an interpreter pays off. In longer running cases the overhead of the compiled cases is over-shadowed by increased times of interpretive execution. For DO loop equivalents where the number of iterations is in the range of 100 to 200, *APL* is faster than FORTRAN and within the scope of the workspace sizes and accuracies available *APL* is in the extreme, from 2 to 10 times slower than the GO step of compiled FORTRAN. In fact such a comparison is too severe. Since any interactive system does scheduling and swapping and portions a share to each process we should also have to count similar amounts when examining the compiled code. Thus we should calculate

$$\begin{aligned}
 & (\text{time schedule compiler} \\
 & \quad + \text{time to compile} \\
 & \quad + \text{time to schedule} \\
 & \qquad \text{Linkage Editor} \\
 & \quad + \text{time to execute the} \\
 & \qquad \text{Linkage Editor} \\
 & \quad + \text{time to schedule} \\
 & \qquad \text{the GO step} \\
 & \quad + N \times \text{GO step time}) \div N
 \end{aligned}$$

where N is positive number giving a measurement of frequency of use.

Such a formula is more equitable but really only gives a reasonable picture when the N runs are sequential, otherwise the scheduler times for the GO step and perhaps the linkage editor step should be apportioned differently.

Relative to FORTRAN coding, particularly in those areas where increased accuracy may be of value but not necessarily needed, programmers should consider using long (double) precision. Neither the time nor the space penalty is commensurate with improved accuracy and not having to worry about conversion problems when mixing precisions.

4.2 Space Requirements

Table 4.2 gives space considerations for the same cases examined in Table 4.1. In APL we give sizes for the space required by the codestring when typed in from the terminal and when the same string is line 1 of a result returning function. Function definition overhead for APL\360 is about 40 bytes plus 8 bytes overhead per line. The word "about" relates to the variability that occurs in the variety of function types, local variables and when the entries in the workspace end on a full word boundary.

The APL codestring sizes are roughly one tenth of the size of

TABLE 4.2
PRIMITIVE CONSTRUCTS

SPACE REQUIREMENTS				
	PROGRAM SIZE (BYTES)		RUN TIME PACKAGE (DYNAMIC DATA SIZE FOR APL)	
	<i>APL</i>	FORTTRAN	<i>APL</i>	FORTTRAN
1) $+/\iota 1$	24 bytes	210	36	20,592
	(codestring)	262		20,640
	68 bytes	262		20,640
	(function)			
2) $+/\iota 2000$		202	8032	20,584
	24 codestring	254		20,622
	68 function	262		20,640
3) $+/2000\rho 1$		214	284	20,592
	32(codestring)	214		20,592
	72 function	222		20,600
4) $+/\iota 7500$		202	30032	20,584
	24(codestring)	254		20,622
	68(function)	262		20,640
5) $\times/\iota 1$		210	36	20,592
	24(codestring)	262		20,640
	68(function)	262		20,640
6) $\times/\iota 56$		270	260	20,648
	24(codestring)	254		20,632
	68(function)	262		20,640

TABLE 4.2 (continued)

	APL	FORTTRAN	APL	FORTTRAN
7) $\times/2000\rho 1$		214	284	20,592
	32(codestring)	214		20,592
	72(function)	222		20,600
8) $+/(11000)\epsilon 11000$		228	8192	20,608
$1 \leq J \leq I$	32(codestring)	240		20,616
	76(function)	240		20,616
$1 \leq J \leq 1000$		238		20,616
		250		20,632
		250		20,632
9) $D+. LD \leftarrow 3 \ 3\rho 19$		396	168	20,776
	44(codestring)	386		20,768
	88(function)	458		20,840
10) $D+. LD \leftarrow 5 \ 5 \ 5\rho 1125$		1078	1572	21,456
	48(codestring)	1076		21,456
	92(function)	1606		21,986

the FORTRAN programs. The overhead penalty for data for *APL* is somewhat higher due to the dynamic nature of the storage of values.

The nature of the interpreter and the data representation also account for the expansion of storage requirements during execution for *APL*\360. The size of the FORTRAN run time package is quite large compared to the program (almost two orders of magnitude). While larger FORTRAN programs are not likely to show as badly, it should be kept in mind that if the average size of a FORTRAN run time package were 20K bytes then 4 FORTRAN programs would carry along requirements of space which in combination would be almost as large as the *APL*\360 interpreter.

4.3 Scalar Functions Extended to Vectors

If any of the advantages of *APL* (the interpreter environment) are to be gained then the strong points of the language based in the interpreter must be exploited. It was decided to examine the use of the extension of the primitive dyadic scalar $+$ to vectors and matrices rather than the use of FORTRAN style looping in *APL*. The object was to gain insight into the cost of the looping and its associated interpretation costs in *APL*. To accomplish this times for the primitive $+$ were measured against the function *ADD* for the vector lengths of 1, 2, 4, 10, 16, 20, 24, 28, 32, 64, 128, 256, 512 and 1024 elements.


```

          VZ ← A ADD B; I
[1]      Z ← (ρA) ρ 0
[2]      I ← 1
[3]      L1: Z[I] ← A[I] + B[I]
[4]      →((I←I+1) ≤ ρA) / L1

```

▽

Clearly *ADD* simulates a FORTRAN-like way of performing vector addition.

Using the *APL* function, Domino (), least square fits of

degree 1 to 5 were made for both the primitive + and the function ADD.

Since the number of loops in ADD or embedded in + is linear, we should expect an adequate fit using the form

$$y_i = a_0 + a_1 \times x_i$$

The results of the least squares fit for polynomials of degrees one and two are summarized in Table 4.3.

DEGREE		OF		POLYNOMIAL	
		1		2	
		+	ADD	+	ADD
a ₀	1.479	1.906		1.628	1.888
a ₁	0.0106	2.184		7.321E ⁻³	2.185
a ₂	-	-		3.490E ⁻⁶	1.111E ⁻⁵
sum of squares	1.302	16.53		0.3583	16.53

TABLE 4.3

The size of the coefficient of the quadratic term relative to the first order coefficient indicates that we will have about 3% difference in what would have been predicted using the linear model when 1000 element arguments are used. The reduction in the sum of squares between the model and actual measurements for + when going from a linear to a quadratic fit is due to the short time needed for execution of the + functions; greater inaccuracies in measurement exist when adding small vectors with numbers of elements and a higher order polynomial fits the dispersed data better.

We conclude that the linear model will be good enough to give reasonable insight into a comparison of the primitive extended to vectors and a FORTRAN-like program simulating the extension.

Examination of the constant coefficients (1.479 and 1.906) would tend to indicate that about 29% more time is required for

initialization in the looping case; however, it should be noted that the *ADD* coding appeared as a function call and thus required interpretation and elaboration above and beyond that which would be needed if the same code appeared in line. The linear terms (0.0106 and 2.184) clearly indicate that simulating the extension is 206 times less efficient than using the primitive. This extra time arises from two sources, the first of which is interpreting the line (or lines) $n - 1$ times more than would be required if looping did not have to be used. In addition to the lines being longer to do the same amount of work, generally two lines are required; one to do the branching and another in which the function is performed with suitable indexing of the vector arguments. It is the use of *APL*'s very general indexing in this oversimplified fashion which adds additional inefficiency not found in the + primitive's accessing the data.

4.4 Scalar Functions Extended to Matrices

When attempting to model the application of a dyadic scalar primitive to rank 2 arrays there are two ways to proceed. One way is to ravel the arguments, use a function having the form of *ADD* from Section 4.3 to perform the scalar dyadic function and then reshape the result. Although this is in effect what *APL* does, we chose to simulate the primitive applied to a matrix in a FORTRAN-like manner, by nested loops. The reason for adopting this approach was to try to get additional insight into the overhead of repetitive looping in *APL*. For square matrices we would expect strong correlation to between the quadratic term of an approximating polynomial in this case and the linear component in the preceeding case. To carry out this investigation matrices of the form

$\nabla MN \leftarrow GEN\ N$

[1] $MN \leftarrow (N,N)_{\rho 1} \ N \times N$

∇

were generated for $N = 1, 2, 4, 10, 16, 20, 24, 28, 32$ and 36 .
Each of these was then added to itself by using the function *MADD*

$\nabla Z \leftarrow A\ MADD\ B; I; J$

[1] $Z \leftarrow (\rho A)_{\rho 0}$

[2] $I \leftarrow 1$

[3] $L1: J \leftarrow 1$

[4] $L2: Z[I;J] \leftarrow A[I;J] + B[I;J]$

[5] $\rightarrow ((J \leftarrow J+1) \leq 1 + \rho A) / L2$

[6] $\rightarrow ((I \leftarrow I+1) \leq 1 + \rho A) / L1$

∇

Once again DOMINO was used to perform least squares fits to the data for both $+$ and *MADD* for polynomials of the first, second and third degree. The coefficients as well as the sum of squares between the data and the approximating polynomials may be summarized by the following table.

DEGREE OF POLYNOMIAL						
1		2		3		
	$+$	<i>MADD</i>	$+$	<i>MADD</i>	$+$	<i>MADD</i>
a_0	-3.243	-331.9	1.802	2.487	1.985	3.950
a_1	0.623	92.07	-0.1550	1.300	-0.2187	0.4697
a_2	-	-	0.0167	2.779	0.0201	2.855
a_3	-	-	-	-	-4.748E-5	-1.600E-3
sum of squares	156.0	4162E5	7.946	11.36	7.787	8.537

TABLE 4.4

The linear fit is rejected immediately not only because of the poor fit denoted by the large sum of squares but also because the negative intercept is misleading in terms of predictive use of the

model. It does indicate the strong dominance of data points away from the origin requiring a polynomial of higher degree to model the behavior of the functions.

A comparison of the second and third degree fits indicates that the cubic coefficient in the polynomial for + accounts for little in reducing the sum of squares in the least squares approximation. Over the range of interest for n ($1 \leq n \leq 36$) the contribution of the cubic term only approaches the size of the constant term. The third order term plays a larger role in creating a model for *MADD*.

The similarities between a_0 in + with vector and matrix arguments and for *ADD* and *MADD* and the similarities between a_1 for *ADD* and + applied to vectors and a_2 for *MADD* and + applied to matrices, lead us to consider the quadratic approximation for + and *MADD* for matrix addition.

The negative value of coefficient a_1 , for + applied to matrices is worthy of comment. It implies that the slope of the approximating polynomial is negative for $n \leq 4$ and positive for $n > 4$. This probably reflects the inaccuracies of the measurement process for small n .

If we consider the two models, $0.0167x^2 - 0.155x + 1.802$ for + and $2.779x^2 + 1.3x + 2.487$ for *MADD* we would expect behavior for large x to be as the ratio of 2.779 to 0.0167 or about 167 to 1. Yet over the range of fit with $n = 32$ so that $n^2 = 1024$ the two polynomials evaluate to numbers having a ratio of about 208 which agrees closely with the ratios of slopes from the linear model derived in the previous section.

4.5 Summation

Within the scope of simple constructs such as reduction, inner products and extensions of scalar function to vectors and arrays of higher rank, there is evidence that *APL* is competitive with FORTRAN when we restrict the size of the arguments to being small

or at least reasonable with regard to the size of the defacto standard workspace of 36K bytes. To achieve advantage where it exists, coding in *APL* must exploit the array capabilities of the language. In general FORTRAN-like constructs must be reformulated to produce good code for the interpretive environment under study. Replacing looping with array structure, in general, and in the particular cases examined here, may be faster than FORTRAN like coding in *APL* by a couple of orders of magnitude.

For good *APL* code and in simple constructs such as given here *APL* can beat the execute times of FORTRAN and is, in extreme cases, no worse than an order of magnitude slower. In fact speeding *APL* up by a factor of 2 or 3 by techniques which would not show an equivalent gain in compiled code would make interpretation in this context quite comparable with FORTRAN execute times.

APL code is 8 to 10 times more compact although there is a much higher penalty for data because of the dynamic size of data. The size of the runtime package of FORTRAN greatly reduces the severity of such problems when comparing the two.

The times charged to *APL* do carry a proportion of the overhead of supervisory tasks as well as language function such as interpretation and elaboration. These same figures are usually not considered in the same light when judging the batch environment but they must be paid for somewhere. On the other hand, the space taken up by a FORTRAN program provides for the data, but often some space is overlayed and other is in COMMON.

5.0 MATRIX INVERSION AND LEAST SQUARES TECHNIQUES

The second area of consideration is that of matrix inverse techniques. This was prompted because routines for matrix inversion have been of demand and standardized to the extent that a variety of algorithms for that task are usually available

in scientific subroutine libraries for the FORTRAN batch environment. Also, the availability of APL's DOMINO (\boxplus) function in IBM's Program Product APL\360 -OS (5734-XM6) and APL\360 -DOS (5736-XM6) invite comparison both within APL and between APL and FORTRAN. Documentation for DOMINO may be found in papers by M.A. Jenkins [6,7], in which he describes DOMINO. He includes a number of meaningful examples in the IBM Technical Report [6] which were examined and measured on Syracuse University's APL\360 system under SUOS. In addition 3×3 through 12×12 Hilbert matrices and a 6×5 A_{11} matrix from p 139 of a text by J.R. Westlake [8] have been timed and compared to their known inverses.

In addition to these comparisons Domino was compared to its simulation in APL as given in [6]. DMD simulates the dyadic form of \boxplus and MMD the monadic case. To give comparison to DMD and MMD both the Gauss-Jordan, GJINV, and the Gauss-Seidel GSINV algorithms were programmed in APL. Examples of these algorithms in APL may be found in Hellerman [9] on pages 60-62 and 63-64 respectively.

The comparable FORTRAN tests were made with MINV of IBM's Scientific Subroutine Library and which calculates inverses for REAL*4 data. Tests using the double precision version DMINV were initially inconclusive and after consideration of results similar to that previously seen when comparing REAL*4 and REAL*8 execution further consideration was abandoned. In MINV the Gauss-Jordan method is used with the determinant also being calculated.

5.1 Results

Denote the cases by the following APL statements or their equivalent statements with the time in 60's of a second.

- 1) $A \leftarrow 3 \ 3 \ \rho \ 4 \ 8 \ 5 \ 3 \ 9 \ 2 \ 7 \ 10 \ 2$
 $B \leftarrow 105 \ 97 \ 114$
- | | | |
|--|-------|-----------|
| a) $B \boxtimes A$ | 2 | |
| b) $(\boxtimes A) + . \times B$ | 4.2 | |
| c) $(T + . \times B) \boxtimes (T \leftarrow \boxtimes A) + . \times A$ | 4.8 | |
| d) $B \text{ DMD } A$ | 104.6 | |
| e) $(MMD \ A) + . \times B$ | 104.4 | |
| f) $(GJINV \ A) + \times B$ | 54.2 | |
| g) $(T + . \times B) \text{ DMD } (T \leftarrow \boxtimes A) + . \times A$ | 104.8 | |
| h) $(MINV \ A) + . \times B$ | 17 | (751 CLG) |
- (in FORTRAN)
- 2) $B \leftarrow 3 \ 2 \ \rho \ 105 \ 72 \ 97 \ 56 \ 114 \ 87$
 A as before
- | | | |
|--|-------|-----------|
| a) $B \boxtimes A$ | 3 | |
| b) $(\boxtimes A) + . \times B$ | 3.6 | |
| c) $(T + . \times B) \boxtimes (T \leftarrow \boxtimes A) + . \times A$ | 4.2 | |
| d) $B \text{ DMD } A$ | 102.4 | |
| e) $(MMD \ A) + . \times B$ | 104.8 | |
| f) (not used) | | |
| g) $(T + . \times B) \text{ DMD } (T \leftarrow \boxtimes A) + . \times A$ | 99.8 | |
| h) $(MINV \ A) + . \times B$ | 18 | (789 CLG) |
- (in FORTRAN)
- 3) $H3 \leftarrow \div \sim 1 + (13) \circ . + 13$
- | | | |
|-------------------|-------|-----------|
| a) $\boxtimes H3$ | 2.6 | |
| b) $MMD \ H3$ | 107.6 | |
| c) $GJINV \ H3$ | 52.2 | |
| d) $MINV \ H3$ | 17 | (686 CLG) |
- (in FORTRAN)
- 4) $H12 \leftarrow \div \sim 1 + (112) \circ . + 112$
- | | | |
|--------------------|-------|-----------|
| a) $\boxtimes H12$ | 38.4 | |
| b) $MMD \ H12$ | 525.4 | |
| c) $GJINV \ H12$ | 679.8 | |
| d) $MINV \ H12$ | 50 | (769 CLG) |
- (in FORTRAN)

5) $M \leftarrow \begin{bmatrix} 6 & 6 & p & 1 & 0 & 0 & 0 & 0 & 1 \\ & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ & & -1 & 1 & 1 & 0 & 0 & 1 \\ & & & 1 & -1 & 1 & 1 & 0 & 1 \\ & & & & -1 & 1 & -1 & 1 & 1 & 1 \\ & & & & & 1 & -1 & 1 & -1 & 1 & -1 \end{bmatrix}$

a) \mathbb{M}	7.2	
b) $MMD\ M$	199.0	
c) $GINV\ M$	173.2	
d) $MINV\ M$	26	(707 CLG)

(in FORTRAN)

In each of the cases where we refer to the FORTRAN figures CLG stands for Compile Load and Go.

If we compare (a), (*APL* times for the monadic use of \mathbb{M}), and (d) (FORTRAN MINV times) for cases 3, 5, 4 as sample points for the inversion of matrices of order 3, 6 and 12 we see that *APL* out performs compiled FORTRAN. The trend of the data appears that at some point the *APL* times will exceed those of FORTRAN. If we fit quadratic equations to both sets of times in order to get a rough idea of the form of the function, we find that *APL* times would be approximated by

$$0.4074 n^2 - 2.133 n + 5.333$$

while the FORTRAN times follow the form of

$$0.1111 n^2 + 2n + 10.$$

The *APL* predicted (and measured) times agree closely with the times reported by Jenkins [7] (p. 384), and based on solution of the difference of the two approximations the cross over point is about $n = 15$.

Jenkins also notes in [7] that for matrices of order greater than 15 DOMINO runs faster in *APL* than the matrix multiplication of two matrices of the same order.

It should be noted that these estimations are based on quadratic fits while in general we expect matrix inversion routines to have run times which are proportional to cubic functions of

the rank of the matrix. While the number of multiplications (and divisions) and additions grows cubically, the other forms of overhead such as the number of times which the looping routines are called grows quadratically. These approximations then can only give an indication of how the relative overheads behave.

The size of the FORTRAN program sizes and load module sizes for each of the pertinent cases are

CASE	PROGRAM SIZE (bytes)	LOAD MODULE SIZE (bytes)
1	396	22,864
2	454	22,920
3	274	22,744
4	1044	23,512
5	412	22,880

The APL functions *GJINV* and *GSINV* require 488 and 364 bytes respectively. The APL function *DMD*, *MMD*, and *LS* which are used to simulate \boxplus require a total of 1804 bytes.

The FORTRAN load module sizes given above include 22,468 bytes for 10 FORTRAN routines including MINV from the Scientific Subroutine Package.

We have not made mention of the APL function *INV* (or *JINV*) found in 1 *ADVANCEDEX* on the APL\360 system. Jenkins figures[6,7] compare that routine to \boxplus and we do not repeat the results here, except to say that the results are roughly comparable to those obtained for *GJINV* and \boxplus .

In terms of the added function of least squares techniques available in \boxplus and *DMD*, *MMD*, and *LS* we note that for

```
AA ← 5 2 ρ 1 1 1 2 1 3 1 4 1 5
BB ← 1.999 3.002 4.001 4.999 5.998
```

we have the following times (in 60's of a second)

$BB \otimes AA$	2
$BB \text{ DMD } AA$	78.4
$(\otimes AA) + . \times BB$	3.8
$(MMD \text{ } AA) + . \times BB$	83.4
$(T+ . \times BB) \otimes (T \leftarrow \otimes AA) + . \times AA$	3.6
$(T+ . \times BB) \text{ DMD } (T \leftarrow \otimes AA) + . \times AA$	76.2

No least squares techniques coding for FORTRAN was produced. When considering the use of iterative techniques like the Gauss-Seidel method, we consider

$W \leftarrow 4 \ 4 \rho \ 11 \ 2 \ 3 \ 4 \ 2 \ 13 \ 4 \ 5 \ 3$
 $\qquad\qquad\qquad 4 \ 15 \ 6 \ 4 \ 5 \ 6 \ 17$
 $R \leftarrow 1 \ 1 \ 1 \ 1 \ 1$

	Times in 60's of a second
$R \otimes W$	3.4
$R \text{ DMD } W$	149.4
$R \text{ GSINV } W$	389.4
(14 iterations)	
$(\otimes W) + . \times R$	6.2
$(MMD \text{ } W) + . \times R$	155.4
$(GJINV \text{ } W) + . \times R$	102
$(T+ . \times R) \otimes (T \leftarrow \otimes W) + . \times W$	6.8
$(T+ . \times R) \text{ DMD } (T \leftarrow \otimes W) + . \times W$	156
$(T+ . \times R) \text{ GSINV } (T \leftarrow \otimes W) + . \times W$	1041.4
(38 iterations)	

No FORTRAN coding corresponding to the Gauss-Seidel method (*GSINV*) was produced; comparison times using *GJINV* are shown, since that is the technique comparable to MINV.

5.2 Summary

From the above we may conclude, as Jenkins did, that DOMINO is much faster (and more accurate) than the matrix inverse routines written in *APL*. When solving linear equations (or systems of

equations) having the form

$$AX = Y$$

in traditional matrix notation, you should perform $X \leftarrow Y \oslash A$ rather than

$$A^{-1}Y$$

as expressed in the form

$$X \leftarrow (\oslash A) +. \times Y.$$

That is, never use the monadic form when the dyadic use is intended.

For matrices of size less than 15×15 , even using the monadic form of DOMINO the, time to invert a matrix is less than the time to execute a comparable program written in FORTRAN H, OPT = 2. When the times to compile and load and go are considered, DOMINO becomes even more competitive. We do not attempt to say how much more competitive because that would depend on how many matrices are inverted when a routine is compiled, scheduled, and executed. That depends on the application or more correctly a broad sample of applications.

In terms of size the codestring $Z \leftarrow B \oslash A$ takes up about 24 bytes and a dyadic function with the above as the definition would take up 64 bytes. This compares to some 400 or so bytes for the FORTRAN program. The load module size should of course be compared to the some 88,000 bytes required by the APL interpreter a small portion of which is of course the code for DOMINO.

6.0 CLOSED PARTITIONS ON THE STATES OF FINITE STATE MACHINES

A partition, π , on the set of states of a finite state machine,

$M = (S, I, O, \delta, \lambda, s_0)$, is a collection of disjoint subsets (blocks) of the set of states, S , whose set union is S . A partition is said to be closed, or have the Substitution Property (SP), if and only if for each input $a \in I$, the set of inputs, maps blocks of π into blocks of π . That is,

$$E_{\pi}(s) = B_{\pi}(t) \rightarrow B_{\pi}(\delta(s, a)) = B_{\pi}(\delta(t, a))$$

so that when states s and t , are in the same block of π then their images under the next-state function, δ , will also be in the same block independent of the input, a . The FORTRAN program which was the initial focal point of this part of the study was written by Thomas F. Piatkowski [10] for interactive use on the Michigan Terminal System at the University of Michigan. This program calculates all partitions, having the substitution property, of a finite state machine which is input interactively as part of the program execution. In addition to the closed partitions enough information is generated in the output to construct the lattice of closed partitions for that machine. Each partition is given together with an identifying number, a measure of its "height" in the lattice and the type of the point according to whether that closed partition is a lattice atom, a basic generator, a two-state generator, or none of these types. A collection *APL* functions to perform these same tasks have been programmed by one of the authors (GHF) and reported upon elsewhere [11]. The *APL* functions are given here as Figure 6.1 and are as they appeared in [11]. Modularity of the functions are as shown because some functions were used with yet other applications dealing with finite state sequential machines. Since that publication the coding has been improved, but the times and sizes reported here

```

VINITIALIZE[0]V
V INITIALIZE;K;T;SV
[1] 'NUMBER OF STATES, N'
[2] N=1
[3] 'NUMBER OF INPUTS, P'
[4] P=1
[5] STATE=1;SV=0
[6] K=1
[7] 'ENTER ROWS OF THE ' (5+
    '11*SV)+ 'STATEOUTPUT', ' TABLE AS
    REQUESTED'
[8] K
[9] +11*1P=PT+1
[10] +8,0[1] 'SIZE ERROR RE-ENTER ROW'
[11] STATE=STATE,T
[12] +8*1N2K+K+1
[13] +16*1SV
[14] 'OUTPUT TABLE REQUIRED? (YES, NO)'
[15] +17*1'H'εM
[16] +6*10*SV~SV
[17] +20*1(ρSTATE)*N*P
[18] OUT=(N,P)ρ0
[19] +21
[20] OUT=(N,P)ρ(N*P)+STATE
[21] STATE=(N,P)ρSTATE
[22] PRINT
V
VSP[0]V
V SP;IJ;I;J;TM;CC;G2;N2;K;B;T;L;Q;SQ
[1] COLS
[2] G2=(N2,N)+((N2+2)N,1)ρ0
[3] K=1
[4] L1:G2[K;IJ[K;]]+1
[5] +L1*1N2K+K+1
[6] K=1
[7] L2:B+1
[8] L3:T+STATE[G2[K;]=B]/IN;
[9] L=1
[10] L4:+L5*1V/0*Q+G2[K;T;L]
[11] +L6,G2[K;T;L]+1+1/G2[K;]
[12] L5:+L6*1((ρQ)=ρSQ)^^/SQε1+SQ+(Q=0)/Q
[13] L7:G2[K;((Q=0)/T;L)],(G2[K;]εSQ)/IN)-B*1/SQ
[14] G2[K;]+G2[K;]-Q\+/(Q-G2[K;]=0)/G2[K;])ε.(SQ=B)/SQ
[15] +L2
[16] L6:+L4*1P2L+L+1
[17] +L3*1(1/G2[K;])εB+B+1
[18] +L2*1N2K+K+1
[19] K=1
[20] L8:G2[K;]+NORMALIZE G2[K;]
[21] +L8*1N2K+K+1
[22] G2+G2[1/G2;]
[23] B+G2*1=Q2
[24] COMPRESS
[25] PP+1N
[26] LEVEL+L+0
[27] L10:Q+(SQ=0=V/B)/1+ρB-ORDER G2
[28] PP+PP, SQ/G2
[29] LEVEL+LEVEL,(+/SQ)ρL+L+1
[30] +L14*11=ρQ
[31] I=1
[32] L11:J+I+1
[33] L12:+L13*1V/G2*1=TM+G2[Q[I;] SUM G2[Q[J;]]
[34] G2+(1 0 +ρG2)ρ(G2),T
[35] L13:+L12*1(ρQ)εJ+J+1
[36] +L11*1(ρQ)εI+I+1
[37] L14:G2+(-(1+ρG2)εQ)/G2
[38] +L10*10<×/ρG2
[39] PP+((ρPP)N),N)ρPP
[40] K=0
[41] L15:K; ' ' ;LEVEL[1+K]; ' ' ;PRT 1+K
[42] +L15*(1+ρPP)>K+K+1
V

```

```

VCOLS[0]V
V COLS;TM;CC
[1] IJ=Q(2,0.5*ρIJ)ρIJ+(I+TM/,QCC),J+(TM+,IN+.<IN)/,CC-(N,N)ρIN+1N
V
VNORMALIZE[0]V
V COMPRESS[0]V
V ORDER[0]V

```

```

V S=NORMALIZE V;K;P;Q;T;IN
[1] S+(ρV)ρK+1
[2] P=1+Q+IN+1ρV
[3] S[T+(VεV[1+Q])/IN]+P
[4] P+P+1
[5] Q+(-QεT)/Q
[6] +3*10<ρQ
V
V COVER[0]V
V SUM[0]V
V U[0]V

```

```

V S=X COVER I;R;T;Q;K
[1] R+1/X[I;]
[2] S+I*1(K+1)+ρX
[3] +5*11=ρT+(X[I;]=K)/1~1+ρX
[4] S+SA^/Q=1+Q+X[I;T]
[5] +3*1R2K+K+1
V
V PRT[0]V
V R=I SUM J;K;B;C;IN
[1] +0*1(ρI+I)ρJ+J
[2] IN+1ρR+(ρI)ρ0
[3] K=1
[4] S1:B+((IεI[K])/IN) U(JεJ[K])/IN
[5] S2:C+B U((IεI[B])/IN) U(JεJ[B])/IN
[6] +S3*1(A/CεB)^^/BεC
[7] +S2,B+C
[8] S3:R[B]+X
[9] +S1*1(ρR)>K+R+0
[10] R=NORMALIZE R
V
V PRINT[0]V
V T=PRINT;C;IN;IP
[1] T+(5*P)ρ 1 0 0 0 0) \ALPH1[1+STATE]
[2] T[:2+0+5*1+JP+(P)+1, '
[3] T[:3+0]~ALPH2[1+OUT]
[4] T+(- 2 3 +ρT)+T
[5] T[1;1+5*IP]~ALPH2[2+IP]
[6] T[2+IN;1]~ALPH1[1+IN+IN]
[7] T[3;]~1
[8] T[2;]~1
V

```

```

V Z=PRT K;A;B;C;IN
[1] C+1+IN+1N
[2] Z+10
[3] B+1/PP[K;]
[4] Z+Z, '(,((1,(2*~1+ρA)ρ 0 0 1)\A+ALPH1[1+(PP[K;]=C)/IN]), '
[5] +4*1B2C+C+1
[6] Z+~1+Z
V

```

```

INITIALIZE
NUMBER OF STATES, N
0:
8
NUMBER OF INPUTS, P
0:
2
ENTER ROWS OF THE STATE TABLE AS REQUESTED
1
0:
3 7
2
0:
4 8
3
0:
1 6
4
0:
2 5
5
0:
2 4
6
0:
1 3
7
0:
4 4
8
0:
3 3
OUTPUT TABLE REQUIRED? (YES, NO)
NO
1 2
A [C,- G,-
B [D,- H,-
C [A,- F,-
D [B,- E,-
E [B,- D,-
F [A,- C,-
G [D,- D,-
H [C,- C,-
SP
0 0 (A);(B);(C);(D);(E);(F);(G);(H)
1 1 (A);(B);(C F);(D);(E);(G);(H)
2 1 (A);(B);(C);(D E);(F);(G);(H)
3 1 (A B);(C D);(E F);(G H)
4 2 (A B C D);(E F G H)
5 2 (A);(B);(C F);(D E);(G);(H)
6 3 (A B);(C D E F);(G H)
7 4 (A B C D E F G H)

```

FIG. 6.1
SP FUNCTIONS

are for those functions as shown in Figure 6.1.

The FORTRAN program [10] together with the *APL* documentation [11] were given to another of the authors of this report (HAES) with instructions to start with the FORTRAN program, determine how it worked, get it running on Syracuse University computing facilities, write one or more programs or collection(s) of functions in *APL* to produce results which were, it was hoped, as good as, if not better than, the *APL* functions cited above. Finally, comparisons among the cases: FORTRAN, his *APL* functions and *SP* from Figure 6.1 were to be made.

These efforts are discussed in the next section with the results given in the section following that.

It should be noted that at the time the programmer (HAES) began, he knew neither FORTRAN nor *APL* but he did know ALGOL. Also, it was not trivial to say "get the program running" because between 1967 and 1971 and between the compiler implementation available to Piatkowski on the Model 67 at Michigan and the one available to Spaanenburg on the Model 50 running under SUOS at Syracuse changes had been made in the FORTRAN compiler so that alterations had to be made to WRITE and FORMAT statements in order to get the program to run.

6.1 Translating from FORTRAN to *APL*

In the following an effort is made to enable the reader, who is familiar with the algorithm, to follow the FORTRAN program and the *APL* functions; however, additional background material may be found in Hartmanis and Stearns [12].

Figure 6.2 shows an annotated Flow Chart of the FORTRAN program as it appeared in [10]. In that program TP1 and TP2 are two linear arrays in each of which temporary information on a single partition may be stored. The format for TP1 and TP2 is the same as for a single PP array segment which we consider next.

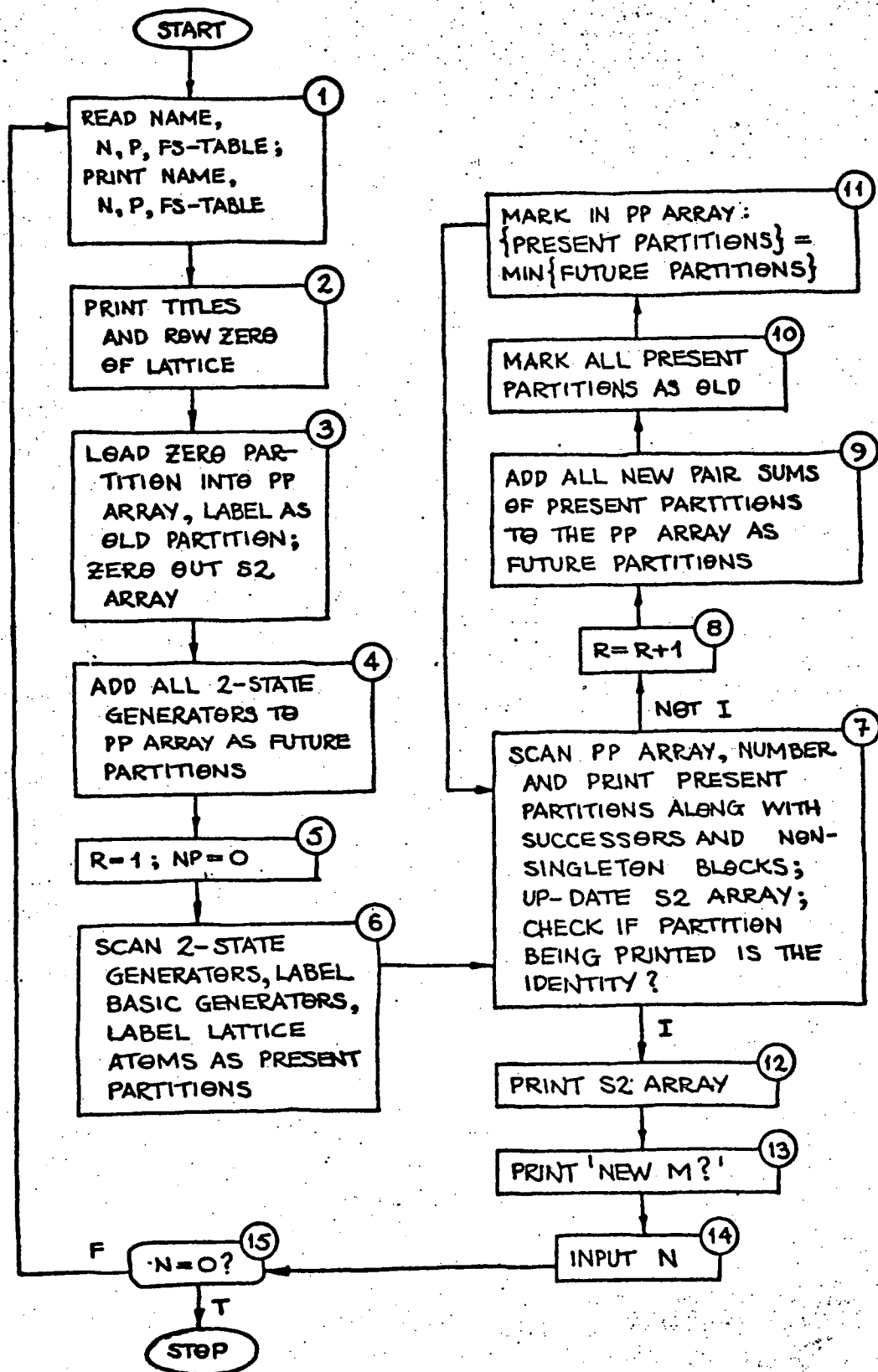
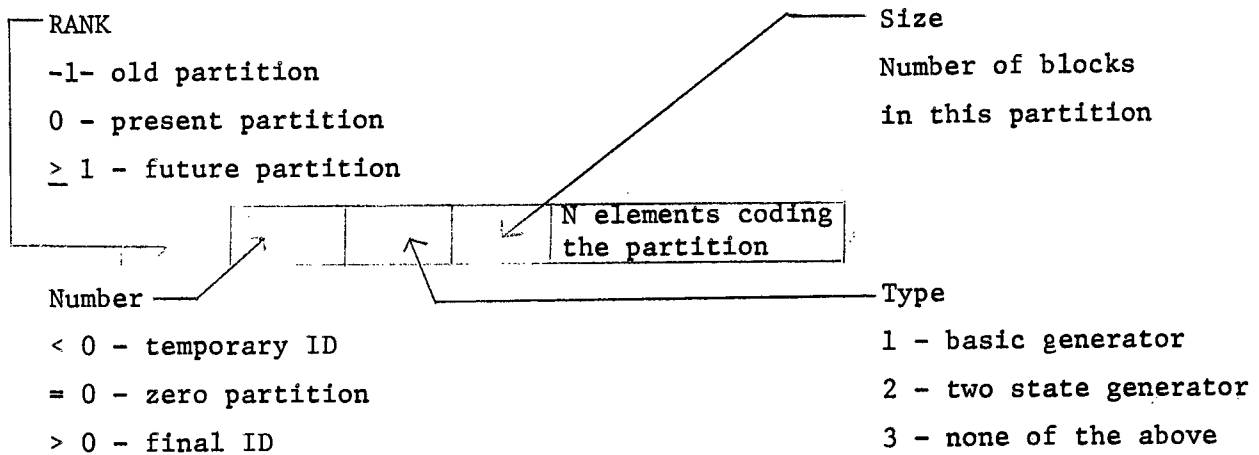


FIG 6.2

FORTTRAN FLOWCHART

PP, in which the permanent partition information is stored, is also a linear array. Each partition occupies a segment of length $N + 4$ in PP where N is the number of states in the machine under consideration. The segment is coded as follows:



cells 5,6 ... $N + 4$ contain coding for the partition. The $i + 4$ th cell marks the block of state i . Two states are in the same block if and only if their cells contain the same number. When the partition is in normal form, cell 5 corresponding to state 1 will contain a 1. The lowest numbered state which is not in the same block as state 1 is marked with 2. The address of the segment corresponds to the location of the $N + 4$ th cell. In APL a normalized partition the number of blocks would be given by $\lceil PP \rceil$ removing the need of SIZE. PPM is the index of the last cell of the last partition in the PP array. One of the philosophic problems is that PP could have been stored as a matrix but keeping PP a vector and being somewhat more independent of N is of value when running a number of problems interactively and in attempting an optimization of allocated storage in the compiler environment. This trade-off slightly complicates the understanding of the program however.

S2 is a two-dimensional array and $S2(I,J)$ is the number (either temporary or final) of the two-state generator partition

obtained by placing states I and J (and only those states) in the same block. If $S2(I,J) = 0$, then the partition is not yet known.

The following subroutines appear in the FORTRAN program and hence play an important role in the *APL* implementation.

SUM(N,TP1,TP2) is a subroutine which places the sum (the lattice function for partitions) of TP1 and TP2 into TP1.

REDUCE(N,P,FS,TP1) is a subroutine which replaces the partition in TP1 with the smallest partition in SP which contains it.

NORSIZ(N,TP1) is a subroutine which normalizes and sizes the partition given in TP1.

EQUAL(N,PPM,TP1,PP,LEQ,PPEQ) is a subroutine which scans the partitions in PP and compares them with the partition in TP1 we set.

$$LEQ \leftarrow \begin{cases} 1 & \text{if a match is found} \\ 0 & \text{otherwise} \end{cases}$$

If there is a match PPEQ in the address of the PP-partition identical to the TP1 partition. All partitions must be normalized and sized.

LESS(J,I,N,PP) is a logical function whose value is .TRUE. if and only if the partition at location J in PP is less than or equal to the partition at location I.

Figure 6.3 which is continued on a number of pages shows both the FORTRAN program and a collection of *APL* functions which comprise the FORTRAN to *APL* translation efforts. The FORTRAN program contains notation along the left margin; the numbers denote segments of the program corresponding to the numbers on the Annotated Flow Chart of Figure 6.2. Located near the appropriate section of the FORTRAN program are (usually) three *APL* functions having the name format of *FNO*, *FN1*, and *FNX*. These are grouped in three groups. The first list in)GRP ZERO

```

)GRPS
FIRST XXXX ZERO
)GRP ZERO
INITIALIZE SP SP10 SP200 SP30 SP40 SP50
SUM REDUCE NORISZ EQUAL LESS PIAT
)GRP FIRST
INITIALIZE SP1 SP11 SP211 SP31 SP41 SP50
SUM1 REDUCE1 NORISZ1 EQUAL1 LESS1 PIAT1
)GRP XXXX
INITIALIZE SPX SP1X SP2XX SP3X SP4X SP50
SUMX REDUCEX NORISZX EQUALX LESSX PIATX

```

```

VPIAT[0]V
V PIAT
[1] INITIALIZE
[2] SP
[3] SP10
[4] SP200
[5] 'NEW MACHINE ( 0=NO , 1=YES )?'
[6] +0
V

```

```

VPIAT1[0]V
V PIAT1
[1] INITIALIZE
[2] SP1
[3] SP11
[4] SP211
[5] 'NEW MACHINE ( 0=NO , 1=YES )?'
[6] +0
V

```

```

VPIATX[0]V
V PIATX
[1] INITIALIZE
[2] SPX
[3] SP1X
[4] SP2XX
[5] 'NEW MACHINE ( 0=NO , 1=YES )?'
[6] +0
V

```

```

VINITIALIZE[0]V
V INITIALIZE
[1] 'SP LATTICE PROGRAM'
[2] 'MACHINE NAME ?'
[3] NAME=0
[4] SP50A: 'NUMBER OF STATES , N '
[5] N=0
[6] +(0<N<101-N)/SP100A
[7] 'N OUT OF RANGE'
[8] +SP50A
[9] SP100A: 'NUMBER OF INPUTS , P'
[10] P=0
[11] +(0<P<6-P)/SP150
[12] 'P OUT OF RANGE'
[13] +SP100A
[14] SP150: 'STATE TRANSITION TABLE:'
[15] 'FOR EACH I ENTER 'I,' NUMBERS (≤N)'
[16] 'CORRESPONDING TO FS[I;J] FOR J=1 TO 'P'
[17] FS+(N,P)PO
[18] I+1
[19] SP200A: 'I= 'I
[20] FS[I;P]+0
[21] +(N+I+1)/SP200A
[22] 'I'
[23] 16P'-
[24] 'I'
[25] 'MACHINE NAME = 'NAME
[26] 'N= 'N; P= 'P
[27] 'I'
[28] 16P'-
[29] 'I'
[30] 'STATE TRANSITION TABLE'
[31] 'I'
[32] 'INPUTS'
[33] 'STATE 'I;P
[34] 'I'
[35] I+1
[36] SP230A: 'I; 'FS[I;P]
[37] +(N+I+1)/SP230A
[38] 'I'
[39] 16P'-
[40] 'I'
[41] 'LATTICE TABLE'
[42] 'CODE: A = LATTICE ATOM'
[43] 'B = BASIC GENERATOR'
[44] '2 = TWO-STATE GENERATOR'
[45] 'I'
[46] 'NO. ROW TYPE'
[47] '0 0 ZERO'
V

```

```

IMPLICIT INTEGER*2(A-Z)
REAL*8 TYPE
LOGICAL LESS
DIMENSION FS(100,5),NAME(50),PP(5000),S2(100,100)
DIMENSION SUCC(100),TP1(104),TP2(104),TYPE(4)
DATA TYPE/'AB2',' B2',' 2',' '/
WRITE(3,10)
10 FORMAT(/19H SP LATTICE PROGRAM)
20 WRITE(3,30)
30 FORMAT(/40H MACHINE NAME?(TYPE UP TO 50 CHARACTERS))
READ(1,40)NAME
40 FORMAT(50A1)
50 WRITE(3,60)
60 FORMAT(/44H N?(TYPE A 3-DIGIT NUMBER IN RANGE 1 TO 100))
READ(1,70)N
70 FORMAT(I3)
IF(N*(101-N))80,80,100
80 WRITE(3,90)
90 FORMAT(17H***N OUT OF RANGE)
GO TO 50
100 WRITE(3,110)
110 FORMAT(/42H P?(TYPE A 1-DIGIT NUMBER IN RANGE 1 TO 5))
READ(1,120)P
120 FORMAT(I1)
IF(P*(6-P))130,130,150
130 WRITE(3,140)
140 FORMAT(17H***P OUT OF RANGE)
GO TO 100
150 WRITE(3,160)
160 FORMAT(/24H STATE TRANSITION TABLE:)
WRITE(3,170)P
170 FORMAT(16H FOR EACH I TYPE,I,2,16H 3-DIGIT NUMBERS)
WRITE(3,171)
171 FORMAT(24H SEPARATED BY COMMAS AND)
WRITE(3,172)
172 FORMAT(25H CORRESPONDING TO FS(I,J))
WRITE(3,180)P
180 FORMAT(11H FOR J=1 TO,I,3)
DO 200 I=1,N
WRITE(3,190)I
190 FORMAT(/5H I = ,I,3)
200 READ(1,210)(FS(I,J),J=1,P)
210 FORMAT(5(I,3,1X))
WRITE(3,213)
WRITE(3,10)
WRITE(3,211)NAME
211 FORMAT(/16H MACHINE NAME = ,50A1)
WRITE(3,212)N,P
212 FORMAT(/5H N = ,I,3,5X,4HP = ,I,3)
WRITE(3,213)
213 FORMAT(/40(1H-))
WRITE(3,214)
214 FORMAT(/23H STATE TRANSITION TABLE)
WRITE(3,220)(I,I=1,P)
220 FORMAT(/12X,6HINPUTS/6H STATE,2X,5I5)
WRITE(3,221)
221 FORMAT(1H )
DO 230 I=1,N
230 WRITE(3,240)I,(FS(I,J),J=1,P)
240 FORMAT(I4,4X,5I5)
WRITE(3,213)
WRITE(3,250)
250 FORMAT(/14H LATTICE TABLE)
WRITE(3,251)
251 FORMAT(/26H TYPE CODE: A=LATTICE ATOM)
WRITE(3,252)
252 FORMAT(12X,17H=BASIC GENERATOR)
WRITE(3,253)
253 FORMAT(12X,21H2=TWO-STATE GENERATOR)
WRITE(3,260)
260 FORMAT(/15H NO. ROW TYPE)
WRITE(3,270)
270 FORMAT(/15H 0 0 ZERO)

```

FIG. 6.3
TRANSLATION STEPS
{33}

```

VSP[ ]V
V SP
[1] N3+N+3
[2] N4+N+4
[3] PPM+N4
[4] PN+1
[5] PP+N4p0
[6] S2*(N,N)p0
[7] PP[1]+1
[8] PP[2]+0
[9] PP[3]+3
[10] PP[4]+N
[11] I+1
[12] SP280A:PP[I+4]+I
[13] J+I
[14] SP280B:S2[I,J]+0
[15] +(N2J+J+1)/SP280B
[16] +(N2I+I+1)/SP280A
[17] TP1+N4p0
[18] TP2+N4p0
[19] I+1
[20] SP400A:J+I
[21] SP400B:+(I=J)/SP400
[22] K+1
[23] SP290A:TP1[K+4]+K
[24] +(N2K+K+1)/SP290A
[25] TP1[J+4]+I
[26] K+1
[27] SP370A:+(FS[I,K]<FS[J,K])/SP298
[28] +(FS[I,K]=FS[J,K])/SP300
[29] S2T+S2[FS[J,K];FS[I,K]]
[30] +SP299
[31] SP298:S2T+S2[FS[I,K];FS[J,K]]
[32] S2T99:+(S2T=0)/SP320
[33] SP300:M+1
[34] SP310A:TP2[M+4]+M
[35] +(N2M+M+1)/SP310A
[36] TP2[FS[J,K]+4]+FS[I,K]
[37] +SP360
[38] SP320:M+2
[39] SP340A:+(PP[M]=S2T)/SP340
[40] MT=M-2
[41] +SP350
[42] SP340:+(PPM2M+M+N4)/SP340A
[43] SP350:M+5
[44] SP355A:TP2[M]+PP[MT+N]
[45] +(N42M+M+1)/SP355A
[46] SP360:SUM
[47] +(P2K+K+1)/SP370A
[48] REDUCE
[49] NORSIZ
[50] EQUAL
[51] +(LEQ<0)/SP390
[52] S2[I,J]+PP[PPEQ-N+2]
[53] +SP400
[54] SP390:PP+PP,N4p0
[55] K+4
[56] SP395A:PP[PPM+K]+TP1[K]
[57] +(N42K+K+1)/SP395A
[58] PP[PPM+3]+2
[59] PP[PPH+2]+PN
[60] PP[PPH+1]+0
[61] S2[I,J]+PN
[62] PH+PH-1
[63] PPM+PPM+N4
[64] SP400:+(N2J+J+1)/SP400B
[65] +(N2I+I+1)/SP400A
V

```

```

VSP10[ ]V
V SP10
[1] R+1
[2] NP+0
[3] N2+8+2*N
[4] I+N2
[5] SP470A:J+1
[6] SP405A:TP1[J+4]+J
[7] +(N2J+J+1)/SP405A
[8] S+0
[9] J+N2
[10] SP430A:+(J=I)VP[PP[I-N]2PP[J-N]]/SP430
[11] +(-J LESS I)/SP430
[12] S+1
[13] JT+J-N4
[14] K+5
[15] SP420A:TP2[K]+PP[JT+K]
[16] +(N42K+K+1)/SP420A
[17] SUM
[18] SP430:+(PPM2J+J+N4)/SP430A
[19] +(S=0)/SP450
[20] NORSIZ
[21] PP[I-N+3]+1
[22] +(TP1[4]=PP[I-N])/SP470
[23] SP450:PP[I-N+1]+1
[24] SP470:+(PPM2I+I+N4)/SP470A
V

```

```

271 PP(1)=1
PP(2)=0
PP(3)=3
PP(4)=N
N3=N+3
N4=N+4
PPM=N4
PN=-1
DO 280 I=1,N
PP(I+4)=I
DO 280 J=1,N
S2(I,J)=0
DO 400 I=1,N
DO 400 J=1,N
IF(I.EQ.J) GO TO 400
DO 290 K=1,N
TP1(K+4)=K
TP1(J+4)=I
DO 370 K=1,P
IF(FS(I,K)-FS(J,K)) 298,300,297
297 S2T=S2(FS(J,K),FS(I,K))
GO TO 299
298 S2T=S2(FS(I,K),FS(J,K))
299 IF(S2T) 320,300,320
300 DO 310 M=1,N
310 TP2(M+4)=M
TP2(FS(J,K)+4)=FS(I,K)
GO TO 360
320 DO 340 M=2,PPM,N4
IF(PP(M)-S2T) 340,330,340
330 MT=M-2
GO TO 350
340 CONTINUE
350 DO 355 M=5,N4
355 TP2(M)=PP(MT+M)
360 CALL SUM(N,TP1,TP2)
370 CONTINUE
CALL REDUCE(N,P,FS,TP1)
CALL NORSIZ(N,TP1)
CALL EQUAL(N,PPM,TP1,PP,LEQ,PPEQ)
IF(LEQ) 390,390,380
380 S2(I,J)=PP(PPEQ-N-2)
GO TO 400
390 DO 395 K=4,N4
395 PP(PPM+K)=TP1(K)
PP(PPH+3)=2
PP(PPH+2)=PN
PP(PPH+1)=0
S2(I,J)=PN
PN=PN-1
PPM=PPM+N4
400 CONTINUE

```

```

R=1
NP=0
N2=2*N+8
DO 470 I=N2,PPM,N4
DO 405 J=1,N
405 TP1(J+4)=J
S=0
DO 430 J=N2,PPM,N4
IF(J.EQ.I.OR.PP(I-N).GE.PP(J-N)) GO TO 430
IF(.NOT.(LESS(J,I,N,PP))) GO TO 430
S=1
JT=J-N4
DO 420 K=5,N4
420 TP2(K)=PP(JT+K)
CALL SUM(N,TP1,TP2)
430 CONTINUE
IF(S) 440,450,440
440 CALL NORSIZ(N,TP1)
PP(I-N-3)=-1
IF(TP1(4)-PP(I-N)) 450,470,450
450 PP(I-N-1)=1
470 CONTINUE

```

FIG. 6.3
CONTINUED
{34}

```

VSP200[ ]V
V SP200
[1] SP471:I+1
[2] SP641A:-(PP[I]#0)/SP641
[3] NP=NP+1
[4] S=PP[I+1]
[5] J+1
[6] SP500A:K+J
[7] SP500B:-(S2[J;K]*S)/SP500
[8] S2[J;K]=NP
[9] SP500:-(N2K+K+1)/SP500B
[10] -(N2J+J+1)/SP500A
[11] PP[I+1]=NP
[12] IT=I+N3
[13] SUCC=0p0
[14] J+1
[15] SP510A:-(PP[J]=1)/SP510
[16] JT=J+N3
[17] -(~JT LESS IT)/SP510
[18] PP[J]=2
[19] SP510:-(PPN2J+J+N4)/SP510A
[20] J+1
[21] SP530A:-(PP[J]=2)/SP530
[22] JT=J+N3
[23] K+1
[24] SP520A:-(PP[K]=2)*K=J)/SP520
[25] KT=K+N3
[26] -(JT LESS KT)/SP525
[27] SP520:-(PPN2K+K+N4)/SP520A
[28] SUCC=SUCC,PP[J+1]
[29] -SP530
[30] SP525:PP[J]+1
[31] SP530:-(PPN2J+J+N4)/SP530A
[32] J+1
[33] SP535A:-(PP[J]=2)/SP535
[34] PP[J]+1
[35] SP535:-(PPN2J+J+N4)/SP535A
[36] -(R+1)/SP550
[37] T+1
[38] -SP600
[39] SP550:-(PP[I+2]=1)/SP570
[40] T+2
[41] -SP600
[42] SP570:-(PP[I+2]=2)/SP590
[43] T+3
[44] -SP600
[45] SP590:T+4
[46] SP600: ' ',NP; ' ',R; ' ',AB2 B2 2 ' '
      [(3*T-1)+1;3]; ' ' SUCC= ' ',SUCC
[47] J=PP[I+3]
[48] J+1
[49] SP640A:SUCC=0pS+0
[50] K+1
[51] SP630A:-(PP[I+3+K]=J)/SP630
[52] S=C+1
[53] SUCC=SUCC,K
[54] SP630:-(N2K+K+1)/SP630A
[55] -(S21)/SP640
[56] ' ', ' ' BLOCK ' ',J; ' ',SUCC
[57] SP640:-(JP2J+J+1)/SP640A
[58] -(PP[I+3]=1)/SP840
[59] SP641:-(PPN2I+I+N4)/SP641A
[60] SP30
[61] SP40
[62] -SP471
[63] SP840:SP50
V

```

```

VSP30[ ]V
V SP30
[1] R=R+1
[2] I+1
[3] SP760A:-(PP[I]#0)/SP760
[4] J+I
[5] SP759A:-(I=J)*PP[J]#0)/SP759
[6] K+4
[7] SP720A:TP1[K+1]+PP[I+K]
[8] TP2[K+1]+PP[J+K]
[9] -(N32K+K+1)/SP720A
[10] SUM
[11] NORSIZ
[12] EQUAL
[13] -(LEQ=0)/SP759
[14] PP+PP,N4p0
[15] K+4
[16] SP750A:PP[PPN+K]+TP1[K]
[17] -(N42K+K+1)/SP750A
[18] PP[PPN+1]+-1
[19] PP[PPN+2]+PN
[20] PN+PN-1
[21] PP[PPN+3]+3
[22] PPM+PPN+N4
[23] SP759:-(PPN2J+J+N4)/SP759A
[24] SP760:-(PPN2I+I+N4)/SP760A
V

```

```

VSP40[ ]V
V SP40
[1] I+1
[2] SP761A:-(PP[I]#0)/SP761
[3] PPI[I]+1
[4] SP761:-(PPN2I+I+N4)/SP761A
[5] I+N4
[6] SP830A:-(PP[I-N3]=1)/SP830
[7] J+N4
[8] SP810A:-(PP[J-N3]=1)*I=J)/SP810
[9] -(J LESS I)/SP830
[10] SP810:-(PPN2J+J+N4)/SP810A
[11] PPI-I-N3)+0
[12] SP830:-(PPN2I+I+N4)/SP830A
V

```

```

471 DO 641 I=1,PPM,N4
    IF(PP(I)) 641,480,641
480 NP=NP+1
    S=PP(I+1)
    DO 500 J=1,N
    DO 500 K=J,N
    IF(S2(J,K)-S) 500,490,500
490 S2(J,K)=NP
500 CONTINUE
    PP(I+1)=NP
    IT=I+N3
    S=0
    DO 510 J=1,PPM,N4
    IF(PP(J).NE.1) GO TO 510
    JT=J+N3
    IF(.NOT.LESS(JT,IT,N,PP)) GO TO 510
    PP(J)=2
510 CONTINUE
    DO 530 J=1,PPM,N4
    IF(PP(J).NE.2) GO TO 530
    JT=J+N3
    DO 520 K=1,PPM,N4
    IF(PP(K).NE.2.OR.K.EQ.J) GO TO 520
    KT=K+N3
    IF(.LESS(JT,KT,N,PP)) GO TO 525
520 CONTINUE
    S=S+1
    SUCC(S)=PP(J+1)
    GO TO 530
525 PP(J)=1
530 CONTINUE
    DO 535 J=1,PPM,N4
    IF(PP(J).EQ.2) PP(J)=1
535 CONTINUE
    IF(R-1) 550,540,550
540 T=1
    GO TO 600
550 IF(PP(I+2)-1) 570,560,570
560 T=2
    GO TO 600
570 IF(PP(I+2)-2) 590,580,590
580 T=3
    GO TO 600
590 T=4
600 WRITE(3,601)NP,R,TYPE(T),(SUCC(J),J=1,S)
601 FORMAT(13,15,3X,A3,3X,5HSUCC:,1014,(/21X,1014))
610 JP=PP(I+3)
    DO 640 J=1,JP
    S=0
    DO 630 K=1,N
    IF(PP(I+3+K)-J) 630,620,630
620 S=S+1
    SUCC(S)=K
630 CONTINUE
    IF(S-1) 640,640,635
635 WRITE(3,636)J,(SUCC(K),K=1,S)
636 FORMAT(19X,6HRLDCK ,13,1H:,1014,(/28X,1014))
640 CONTINUE
    IF(PP(I+3).EQ.1) GO TO 840
641 CONTINUE

```

```

642 R=R+1
DO 760 I=1,PPM,N4
IF(PP(I)) 760,700,760
700 IT=I+N3
DO 759 J=1,PPM,N4
IF(I.EQ.J.OR.PP(J).NE.0) GO TO 759
DO 720 K=4,N3
TP1(K+1)=PP(I+K)
720 TP2(K+1)=PP(J+K)
CALL SUM(N,TP1,TP2)
CALL NORSIZ(N,TP1)
CALL EQUAL(N,PPM,TP1,PP,LEQ,PPE0)
IF(LEQ) 759,740,759
740 DO 750 K=4,N4
750 PP(PPM+K)=TP1(K)
PP(PPM+1)=-1
PP(PPM+2)=PN
PN=PN-1
PP(PPM+3)=3
PPM=PPM+N4
759 CONTINUE
760 CONTINUE

```

```

DO 761 I=1,PPM,N4
IF(PP(I).EQ.0) PP(I)=1
761 CONTINUE
DO 830 I=N4,PPM,N4
IF(PP(I-N3).EQ.1) GO TO 830
DO 810 J=N4,PPM,N4
IF(PP(J-N3).EQ.1.OR.I.EQ.J) GO TO 810
IF(.LESS(J,I,N,PP)) GO TO 830
810 CONTINUE
PP(I-N3)=0
830 CONTINUE
GO TO 471

```

FIG. 6.3
CONTINUED
(35)

```

VSP1[ ]V
V SP1
[1] PPH+ $N_4+1+N_3+N+3$ 
[2] PH+1
[3] PF+1,0,3,N,N
[4] S2+(N,N) $\rho_0$ 
[5] TP1+TP2+N $\rho_0$ 
[6] I+1
[7] SP400A:J+I
[8] SP400B:=(I=J)/SP400
[9] TP1+TP1[14],((J-1)+N),I,(J-N)+N
[10] K+1
[11] SP370A:=(FS[I;K]=FS[J;K])/SP300
[12] +(0=S2T+S2[FS[I;K]FS[J;K];FS[I;K]FS[J;K])/SP320
[13] SP300:TP2+TP2[14],((FS[J;K]-1)+N),FS[I;K],(FS[J;K]-N)+N
[14] +SP360
[15] SP320:TP2+TP2[14],PP[4+(N)+HT+N $\times^{-1}+((PPH+N_4),N_4)\rho PP$ ];
2[S2T]
[16] SP360:SUM1
[17] +(P $\geq$ K+1)/SP370A
[18] REDUCE1
[19] NORISIZ1
[20] EQUAL1
[21] +(LEQ=0)/SP390
[22] S2[I;J]+PP[PPEQ-N+2]
[23] +SP400
[24] SP390:PP+PP,0,PN,2,TP1[3+N+1]
[25] S2[I;J]+PH
[26] PH+PH-1
[27] PPH+PPH+N $\rho_4$ 
[28] SP400:=(N $\geq$ J+1)/SP400B
[29] +(N $\geq$ I+1)/SP400A
V

```

VSP11[]V

```

V SP11
[1] HP+~R+1
[2] I+N2+8 $\times$ N
[3] SP470A:TP1+TP1[14],N
[4] S+0
[5] J+N2
[6] SP430A:=(J=I) $\vee$ PP[I-N] $\geq$ PP[J-N]/SP430
[7] +(~J LESS1 I)/SP430
[8] S+1
[9] TP2+TP2[14],PP[J-N-N]
[10] SUM1
[11] SP430:=(PPH $\geq$ J+J+N $\rho_4$ )/SP430A
[12] +(S=0)/SP450
[13] NORISIZ1
[14] PP[I-N+3]+~1
[15] +(TP1[4]=PP[I-N])/SP470
[16] SP450:PP[I-N+1]+1
[17] SP470:=(PPM $\geq$ I+I+N $\rho_4$ )/SP470A
V

```

VSP211[]V

```

V SP211
[1] SP471:I+1
[2] SP641A:=(PP[I]=0)/SP641
[3] S2*(S*PP[I+1]+NP+NP+1)+S2 $\times$ ~S+S2=PF[I+1]
[4] IT+I+N3
[5] SUCC+0 $\rho_0$ 
[6] J+1
[7] SP510A:=(PP[J] $\neq$ 1)/SP510
[8] JT+J+N3
[9] +(~JT LESS1 IT)/SP510
[10] PP[J]+2
[11] SP510:=(PPH $\geq$ J+J+N $\rho_4$ )/SP510A
[12] J+1
[13] SP530A:=(PP[J] $\neq$ 2)/SP530
[14] JT+J+N3
[15] K+1
[16] SP520A:=(PP[K] $\neq$ 2) $\vee$ K=J/SP520
[17] KT+K+N3
[18] +(JT LESS1 KT)/SP525
[19] SP520:=(PPH $\geq$ K+K+N $\rho_4$ )/SP520A
[20] SUCC+SUCC,PP[J+1]
[21] +SP530
[22] SP525:PP[J]+1
[23] SP530:=(PPH $\geq$ J+J+N $\rho_4$ )/SP530A
[24] J+1
[25] SP535A:=(PP[J] $\times$ 2)/SP535
[26] PP[J]+1
[27] SP535:=(PPH $\geq$ J+J+N $\rho_4$ )/SP535A
[28] T+1+((R=1),(PP[I+2]=1 2),1)/14
[29] ' ',NP; ' ',R; ' ',AB2 B2 2 '[(3+T-1)+
3]; ' ',SUCC
[30] J+1
[31] SP640A:=(1=+K+PP[I+3+N] $\epsilon$ J)/SP640
[32] ' ',BLOCK ' ',J; ' ': ' ',K/N
[33] SP640:=(PP[I+3] $\geq$ J+J+1)/SP640A
[34] +(PP[I+3]=1)/SP640
[35] SP641:=(PPH $\geq$ I+I+N $\rho_4$ )/SP641A
[36] SP31
[37] SP41
[38] +SP471
[39] SP840:SP50
V

```

VSPX[]V

V SPX

```

[1] N4+1+N3+N+4+PN+~1
[2] PP+(1,N $\rho_4$ ) $\rho_1$ ,0,3,N,N
[3] S2+(N,N) $\rho$ TP1+TP2+N $\rho_0$ 
[4] I+1
[5] SP400A:J+I+1
[6] SP400B:TP1+(J-1),I,J+N-J
[7] K+1
[8] SP370A:=(0=S2T+S2[FS[I;K]FS[J;K];FS[I;K]FS[J;K])/SP320
[9] TP2+(FS[J;K]-1),FS[I;K],FS[J;K]+N-FS[J;K]
[10] +SP360
[11] SP320:TP2+PP[PP[2];S2T;4+N]
[12] SP360:SUMX
[13] +(P $\geq$ K+1)/SP370A
[14] REDUCEX
[15] NORISIX
[16] EQUALX
[17] +(0=LEQ)/SP390
[18] S2[I;J]+PP[PPEQ;2]
[19] +SP400
[20] SP390:PP+((PN+PN-1),N $\rho_4$ ) $\rho$ (PP),0,(S2[I;J]+PN),2,(I/TP1),TP1
[21] SP400:=(N $\geq$ J+1)/SP400B
[22] +(N-1) $\geq$ I+1/SP400A
V

```

VSP1X[]V

V SP1X

```

[1] I+1+R+~NP+0
[2] SP470A:S+~1+TP1+N
[3] J+2
[4] SP430A:=(PP[I;4] $\geq$ PP[J;4])/SP430
[5] +(J LESSX I)/SP430
[6] S+1+TP2+PP[J;4+N]
[7] SUMX
[8] SP430:=(|PN) $\geq$ J+J+1/SP430A
[9] +(S=0)/SP450
[10] NORISIX
[11] PP[I;1]+~1
[12] +(PP[I;4]=I/TP1)/SP470
[13] SP450:PP[I;3]+1
[14] SP470:=(|PN) $\geq$ I+I+1/SP470A
V

```

VSP2XX[]V

V SP2XX

```

[1] SP471:I+1
[2] SP641A:=(PP[I;1]=0)/SP641
[3] S2*(S*PP[I;2]+NP+NP+1)+S2 $\times$ ~S+S2=PP[I;2]
[4] SUCC+0 $\rho_0$ 
[5] J+1
[6] SP510A:=(PP[J;1] $\neq$ 1)/SP510
[7] +(J LESSX I)/SP510
[8] PP[J;1]+2
[9] SP510:=(|PN) $\geq$ J+J+1/SP510A
[10] J+1
[11] SP530A:=(PP[J;1] $\neq$ 2)/SP530
[12] K+1
[13] SP520A:=(PP[K;1] $\neq$ 2) $\vee$ K=J/SP520
[14] +(~J LESSX K)/SP525
[15] SP520:=(|PN) $\geq$ K+K+1/SP520A
[16] SUCC+SUCC,PP[J;2]
[17] +SP530
[18] SP525:PP[J;1]+1
[19] SP530:=(|PN) $\geq$ J+J+1/SP530A
[20] PP[1;1]+(PP[1;1] $\times$ ~S)+S+PP[1;1]=2
[21] T+1+((R=1),(PP[I;3]=1 2),1)/14
[22] ' ',NP; ' ',R; ' ',AB2 B2 2 '[(3+T-1)+
3]; ' ',SUCC
[23] J+1
[24] SP640A:=(1=+K+PP[I;4+N] $\epsilon$ J)/SP640
[25] ' ',BLOCK ' ',J; ' ': ' ',K/N
[26] SP640:=(PP[I;4] $\geq$ J+J+1)/SP640A
[27] +(PP[I;4]=1)/SP640
[28] SP641:=(|PN) $\geq$ I+I+1/SP641A
[29] SP3X
[30] SP4X
[31] +SP471
[32] SP840:SP50
V

```

```

VSP31[0]V
V SP31
[1] R+R+I+1
[2] SP760A:=(PP[I]*0)/SP760
[3] J+I
[4] SP759A:=((I=J)*VPP[J]*0)/SP759
[5] TP1+TP1[14],PP[I+3+1N]
[6] TP2+TP2[14],PP[J+3+1N]
[7] SUM1
[8] NORSI21
[9] EQUAL1
[10] -(LEQ=0)/SP759
[11] PP+PP,N4*0
[12] PP[PPN+1,N4]*-1,PN,3,TP1[3+1N+1]
[13] PN+PN-1
[14] PPM+PPM+N4
[15] SP759:=(PPM+J+J+N4)/SP759A
[16] SP760:=(PPM+I+I+N4)/SP760A
V

```

```

VSP41[0]V
V SP41
[1] I+1
[2] SP761A:=(PP[I]*0)/SP761
[3] PP[I]+1
[4] SP761:=(PPM+I+I+N4)/SP761A
[5] I=N4
[6] SP830A:=(PP[I-N3]=1)/SP830
[7] J=N4
[8] SP810A:=((PP[J-N3]=1)*I=J)/SP810
[9] +(J LESS1 I)/SP830
[10] SP810:=(PPM+J+J+N4)/SP810A
[11] PP[I-N3]+0
[12] SP830:=(PPM+I+I+N4)/SP830A
V

```

```

VSP50[0]V
V SP50
[1] SP840:''
[2] 16p''-
[3] ''
[4] 'TWO-STATE GENERATOR TABLE'
[5] 'STATE STATE PARTITION NO.'
[6] I=1
[7] SP850A:J+I
[8] SP850B:=(I=J)/SP850
[9] I: '' ;J: '' ;S2[I;J]
[10] SP850:=(N+J+J+1)/SP850B
[11] +(N+I+I+1)/SP850A
[12] ''
[13] 16p''-
[14] ''
V

```

```

PIATX
SP LATTICE PROGRAM
MACHINE NAME ?
PIAT
NUMBER OF STATES , N
0:
8
NUMBER OF INPUTS , P
0:
2
STATE TRANSITION TABLE:
FOR EACH I ENTER 2 NUMBERS (≤N)
CORRESPONDING TO FS[I;J] FOR J=1 TO 2
I= 1
0:
3 7
I= 2
0:
4 8
I= 3
0:
1 6
I= 4
0:
2 5
I= 5
0:
2 4
I= 6
0:
1 3
I= 7
0:
4 4
I= 8
0:
3 3

```

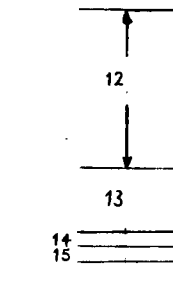
MACHINE NAME = PIAT
N= 8 P= 2

STATE TRANSITION TABLE

STATE	INPUTS
1	3 7
2	4 8
3	1 6
4	2 5
5	2 4
6	1 3
7	4 4
8	3 3

LATTICE TABLE
CODE: A = LATTICE ATOM
B = BASIC GENERATOR
2 = TWO-STATE GENERATOR

NO.	ROW	TYPE
0	0	ZERO
1	1	AB2
2	1	AB2
3	1	AB2
4	2	B2
5	2	
6	3	2
7	4	2



```

VSP3X[0]V
V SP3X
[1] R+R+I+1
[2] SP760A:=(PP[I;1]*0)/SP760
[3] J+I+1
[4] SP759A:=(PP[J;1]*0)/SP759
[5] TP1+PP[I;4+1N]
[6] TP2+PP[J;4+1N]
[7] SUMX
[8] NORSI2X
[9] EQUALX
[10] -(O=LEQ)/SP759
[11] PP+((|PN+PN-1),N4)*0(,PP),-1,PN,3,(I/TP1),TP1
[12] SP759:=(|PN+J+J+1)/SP759A
[13] SP760:=(|PN+1)*I+I+1)/SP760A
V

```

```

VSP4X[0]V
V SP4X
[1] PP[;1]+(PP[;1]*~S)+S+PP[;1]=0
[2] I+1
[3] SP830A:=(PP[I;1]=1)/SP830
[4] J+1
[5] SP810A:=(PP[J;1]=1)*I=J)/SP810
[6] +(J LESSX I)/SP830
[7] SP810:=(|PN)*J+J+1)/SP810A
[8] PP[I;1]+0
[9] SP830:=(|PN)*I+I+1)/SP830A
V

```

```

840 WRITE(3,213)
WRITE(3,841)
841 FORMAT(1/26H TWO-STATE GENERATOR TABLE)
WRITE(3,842)
842 FORMAT(1/5X,25HSTATE STATE PARTITION NO.,/1H )
DO 850 I=1,N
DO 850 J=1,N
IF(1.E0,J) GO TO 850
WRITE(3,851)I,J,S2(I,J)
850 CONTINUE
851 FORMAT(3X,316)
WRITE(3,213)
WRITE(3,852)
852 FORMAT(1/26H NEW MACHINE(0=NO, 1=YES)?)
READ(1,120)N
IF(N) 20,860,20
860 STOP
END

```

TWO-STATE GENERATOR TABLE STATE STATE PARTITION NO.

1	2	1
1	3	4
1	4	4
1	5	7
1	6	7
1	7	7
1	8	7
2	3	4
2	4	4
2	5	7
2	6	7
2	7	7
2	8	7
3	4	1
3	5	6
3	6	2
3	7	7
3	8	7
4	5	3
4	6	6
4	7	7
4	8	7
5	6	1
5	7	4
5	8	4
6	7	4
6	8	4
7	8	1

NEW MACHINE (0=NO , 1=YES) ?

0

consists of the functions obtained by a literal translation of the FORTRAN programs. All of the DO loops in FORTRAN remain as a loop structure in the APL functions. In the places where this leads to obvious misuse of APL corrections are made and the resulting programs are contained in)GRP FIRST. Function names are of the form FN1 here. In this second attempt assignments are also combined. For instance lines 7 through 16 of SP are combined into lines 3 and 4 of SP1 which we would denote by $SP[7], \dots, [16] \rightarrow SP[3] [4]$. In making the transition from those functions grouped in)GRP FIRST to those in)GRP XXXX a matrix representation was used for PP rather than a vector form. This resulted in being able to make use of inner and outer products in manipulating PP such as in $SPX[11]$ and $SP1[15]$. TP1 and TP2 are reduced to contain just the partition and not the coding information. Redundant statements such as $SP1[8]$ and $SP1[11]$ are removed. In GROUP XXXX it is no longer necessary to keep track of PPM and partitions are much easier to address; see $SP1[22] \leftrightarrow SPX[18]$.

As shown in the last page of the Continued Figure 6.3 (p37), the driving functions for each of the three stages in the FORTRAN to APL translation are given by PIAT PIAT1 and PIATX. Figures 6.4 through 6.8 give the various translations of the original FORTRAN subroutines: SUM, REDUCE, NORSIZ, EQUAL and LESS. In most cases by the time the third cut at programming was made the APL functions were down to 1 line. In SUM X a straightforward search is made to find an I such that

$$(TP1 \in TP1 [(TP2 \in TP2[I])/\backslash N])/\backslash N$$

is not empty. This reduces greatly the amount of looping compared to SUM 1, where all indices are found serially. In REDUCE a vector I is again found in a rather straightforward fashion, so that it contains all of the indices necessary to make changes in TP1.

```

VREDUCE[ ]V
V REDUCE;I;J;K;M
[1] RED9:I+1
[2] RED40A:J+1
[3] RED40B:-(TP1[I+4]*TP1[J+4])/RED40
[4] K+1
[5] RED30A:-(TP1[FS[I;K]+4]*TP1[FS[J;K]+4])/RED30
[6] A+TP1[FS[I;K]+4]
[7] B+TP1[FS[J;K]+4]
[8] M+5
[9] RED20A:-(TP1[M]*B)/RED20
[10] TP1[M]*A
[11] RED20:-(N42M+M+1)/RED20A
[12] +RED9
[13] RED30:-(P2K+K+1)/RED30A
[14] RED40:-(N2J+J+1)/RED40B
[15] +(N2I+I+1)/RED40A
V

VREDUCE1[ ]V
V REDUCE1;I;J;K
[1] RED9:I+1
[2] RED40A:J+1
[3] RED40B:-(TP1[I+4]*TP1[J+4])/RED40
[4] K+1
[5] RED30A:-(A+TP1[FS[I;K]+4]*B+TP1[FS[J;K]+4])/RED30
[6] TP1[4+(TP1[4+1N]=B)/1N]*A
[7] +RED9
[8] RED30:-(P2K+K+1)/RED30A
[9] RED40:-(N2J+J+1)/RED40B
[10] +(N2I+I+1)/RED40A
V

VSUNX[ ]V
V SUNX;I;J;K
[1] +1+1+1+(0,0,0)*1+(,0,0,0)*TP1[FS[I;K]+4]*TP1[FS[J;K]+4]*1
[2] +1,TP1[(TP1[X[I;3;K])/1N]*X[I;2;K]+1+(,X[I;2;K]*X[I;3;K])/1N]
V

```

```

SUBROUTINE REDUCE (N,P,FS,TP1)
IMPLICIT INTEGER*2(A-Z)
DIMENSION FS(100,5),TP1(104)
1 N4=N+4
9 CONTINUE
10 DO 40 I=1,N
DO 40 J=1,N
IF(TP1(I+4).NE.TP1(J+4)) GO TO 40
DO 30 K=1,P
IF(TP1(FS(I,K)+4).EQ.TP1(FS(J,K)+4)) GO TO 30
A=TP1(FS(I,K)+4)
B=TP1(FS(J,K)+4)
DO 20 M=5,N4
IF(TP1(M).EQ.B) TP1(M)=A
20 CONTINUE
GO TO 9
30 CONTINUE
40 CONTINUE
50 RETURN
END

```

FIG. 6.4
SUBROUTINE REDUCE

```

VSUM[ ]V
V SUM;I;J;K
[1] I+5
[2] SUM40A:-(TP2[I]=0)/SUM40
[3] A+TP2[I]
[4] J+1
[5] SUM30A:-(TP2[J]*A)/SUM30
[6] +(TP1[I]=TP1[J])/SUM20
[7] B+TP1[I]
[8] C+TP1[J]
[9] K+5
[10] SUM10A:-(TP1[K]=C)/SUM10
[11] TP1[K]*B
[12] SUM10:-(N42K+K+1)/SUM10A
[13] SUM20:TP2[J]+0
[14] SUM30:-(N42J+J+1)/SUM30A
[15] SUM40:-(N42I+I+1)/SUM40A
V

VSUM1[ ]V
V SUM1;I;J
[1] I+5
[2] SUM40A:-(0=A+TP2[I])/SUM40
[3] J+1
[4] SUM30A:-(TP2[J]*A)/SUM30
[5] +((B+TP1[I])=C+TP1[J])/SUM20
[6] TP1[4+(TP1[4+1N]=C)/1N]*B
[7] SUM20:TP2[J]+0
[8] SUM30:-(N42J+J+1)/SUM30A
[9] SUM40:-(N42I+I+1)/SUM40A
V

VSUNX1[ ]V
V SUMX;I;J
[1] +2*xI+1+(V/(TP2[FS(I,K)+4]*TP2[FS(J,K)+4])/1N)*TP1[FS(I,K)+4]*TP1[FS(J,K)+4]
[2] +1,TP1[(TP1[X[I;3;K])/1N]*X[I;2;K]+1+(,X[I;2;K]*X[I;3;K])/1N]
V

```

```

SUBROUTINE SUM (N,TP1,TP2)
IMPLICIT INTEGER*2(A-Z)
DIMENSION TP1(104),TP2(104)
N4=N+4
DO 40 I=5,N4
IF(TP2(I).EQ.0) GO TO 40
A=TP2(I)
DO 30 J=1,N4
IF(TP2(J).NE.A) GO TO 30
IF(TP1(I).EQ.TP1(J)) GO TO 20
B=TP1(I)
C=TP1(J)
DO 10 K=5,N4
IF(TP1(K).EQ.C) TP1(K)=B
10 CONTINUE
20 TP2(J)=0
30 CONTINUE
40 CONTINUE
RETURN
END

```

FIG. 6.5
SUBROUTINE SUM

```

VNORSIZ1[ ]V
V NORSIZ1;I;J;K
[1] I=5
[2] NOR10:TP1[I]+-TP1[J]
[3] +(N4≥I+1)/NOR10
[4] I=1
[5] J=5
[6] NOR30A:+(TP1[J]>0)/NOR30
[7] A=TP1[J]
[8] K=J
[9] NOR20A:+(TP1[K]=A)/NOR20
[10] TP1[K]=I
[11] NOR20:+(N4≥K+1)/NOR20A
[12] I=I+1
[13] NOR30:+(N4≥J+1)/NOR30A
[14] TP1[4]=I-1
V
VNORSIZ1[ ]V
V NORSIZ1;I;J
[1] TP1=TP1[4],-TP1[4+N]
[2] I=1
[3] J=5
[4] NOR30A:+(0<A+TP1[J])/NOR30
[5] TP1[1+J+(TP1[1+J+15+N-J]=A)/15+N-J]-I
[6] I=I+1
[7] NOR30:+(N4≥J+1)/NOR30A
[8] TP1[4]=I-1
V
VNORSIZX[ ]V
V NORSIZX
[1] TP1=((N) * (((N) * . = TP1) [4TP1:1:N;])
V

```

```

SUBROUTINE NORSIZ (N,TP1)
IMPLICIT INTEGER*2(A-Z)
DIMENSION TP1(104)
N4=N+4
DO 10 I=5,N4
10 TP1(I)=-TP1(I)
I=1
DO 30 J=5,N4
IF (TP1(J)) 15,15,30
15 A=TP1(J)
DO 20 K=J,N4
IF (TP1(K).EQ.A) TP1(K)=I
20 CONTINUE
I=I+1
30 CONTINUE
TP1(4)=I-1
RETURN
END

```

FIG. 6.6

SUBROUTINE NORSIZ

```

VEQUAL[ ]V
V EQUAL;I;J
[1] I=N4
[2] EQU20A:J=4
[3] EQU10:+(TP1[J]*PP[I+J-1+4])/EQU20
[4] EQU10:+(N4≥J+1)/EQU10A
[5] LEQ=1
[6] PPEQ=I
[7] +0
[8] EQU20:+(PFM≥I+I+N4)/EQU20A
[9] LEQ=0
V
VEQUAL1[ ]V
V EQUAL1
[1] +(0=LEQ+V/EQ+Λ/(((PFM+N4),N4)P PF=PFM,TP1)[3+1N+1])/0
[2] PPEQ=N4×EQ,1
V
VEQUALX[ ]V
V EQUALX
[1] LEQ+((1PM)≥PPEQ+(PP[4+1N]Λ.=TP1):1
V

```

```

SUBROUTINE EQUAL (N,PPM,TP1,PP,LEQ,PPEQ)
IMPLICIT INTEGER*2(A-Z)
DIMENSION TP1(104),PP(5000)
N4=N+4
DO 20 I=N4,PPM,N4
DO 10 J=4,N4
IF (TP1(J).NE.PP(1-N+4+J)) GO TO 20
10 CONTINUE
LEQ=1
PPEQ=I
GO TO 30
20 CONTINUE
LEQ=0
30 RETURN
END

```

FIG. 6.7

SUBROUTINE EQUAL

```

VLESS[ ]V
V Y+X LESS Z;K;M
[1] ZT=Z-N
[2] XT=X-N
[3] K=1
[4] LESS10A:N+K
[5] LESS10B:+(PP[XT+K]=PP[XT+N])ΛPP[ZT+K]=PP[ZT+N])/LESS20
[6] +(N≥M+1)/LESS10B
[7] +(N≥K+K+1)/LESS10A
[8] Y=1
[9] +0
[10] LESS20:Y=0
V
VLESS1[ ]V
V Y+X LESS1 Z;K
[1] ZT=Z-N
[2] XT=X-N
[3] K=1
[4] LESS10A:+(V/(PP[XT+K]=PP[1+XT+K+1K+1-K])ΛPP[ZT+K]=PP[1+ZT+K+1N+1-K])/LESS20
[5] +(N≥K+1)/LESS10A
[6] Y=1
[7] +0
[8] LESS20:Y=0
V
VLESSX[ ]V
V Y+X LESSX Z
[1] Y=V/V/(PP[X;4+1N]Λ=PP[X;4+1N])ΛPP[Z;4+1N]Λ=PP[Z;4+1N]
V

```

```

LOGICAL FUNCTION LESS(I,I,N,PP)
IMPLICIT INTEGER*2(A-Z)
DIMENSION PP(5000)
I1=I-N
J1=J-N
DO 10 K=1,N
DO 10 M=K,N
IF (PP(I1+K).EQ.PP(I1+M).AND.PP(I1+K).NE.PP(I1+M)) GO TO 20
10 CONTINUE
LESS=.TRUE.
RETURN
20 LESS=.FALSE.
RETURN
END

```

FIG. 6.8

SUBROUTINE LESS

6.2 Results for Time and Space

FORTTRAN

CPUTIME (seconds)	Compiler		
	G	H(Opt=0)	H(Opt=1)
COMPILE	36.89	30.55	43.10
Link Edit	4.25	4.49	3.77
GO	2.75	2.77	2.49
Total Scheduler	4.67	4.92	4.69
Total	48.56	42.73	54.05
Storage (bytes)			
MAIN	38,846	38,416	37,034
SUM	676	646	534
REDUCE	842	790	596
NORSIZ	578	558	454
EQUAL	572	516	450
LESS	660	508	426
TOTAL	62,576	61,832	59,896

APL

CPUTIME (seconds)	Programs			
	ZERO	FIRST	XXXX	SP (GHP)
Execution	592.4	508.4	85.3	62.6
Storage (bytes)				
Before Execution	12360	9324	7564	5116
largest during execution	14884	11556	9448	7680
After Execution	13748	10712	8716	5628

Clearly there is a trade off in time and space between the two modes of operation. If one were to compute the product of space and time using the maximum space in *APL* and the Link Edit, GO and Schedule time, but not the Compile time in FORTTRAN, we have (in byte seconds):

FORTTRAN:	G	H(Opt=0)	H(Opt=1)	
	730,262	753,114	655,861	
APL:	ZERO	FIRST	XXXX	SP
	8,817,282	5,879,693	805,914	480,716

Independent of any value judgements as to what these figures may or may not mean, one lesson which is clear is that if any value is to be gained in the use of *APL* it will require programming in a style which is suited for *APL* and not directly following the programming style found in a FORTRAN program. This can either be done by a re-analysis of the algorithm implementation or by an iterative improvement scheme. In either case computational efficiency can only be gained by using program constructs which are not readily obvious in the FORTRAN-like program.

Re-examination of the critical subroutines in Figs. 6.9 - 6.12 indicates that when translating from *ZERO* to *FIRST* there are instances when the second subroutine runs slower than the original although the averages of the ensemble are less.

The stratification of times, particularly relating to group *XXXX*, denotes that time in execution for the subroutine in question occurs in quanta. These are predictable from examination of the coding.

7.0 THE FAST FOURIER TRANSFORM

The Fourier transform has always been of interest to the scientific community, but the computational efficiencies found in those procedures termed the Fast Fourier Transform (FFT) have recently allowed the Fourier Transform to emerge as an effective problem solving tool [13,14]. Further, the array structure of the procedure appears to lend itself to an *APL* implementation for interactive use.

A FORTRAN H program was written, essentially by translating

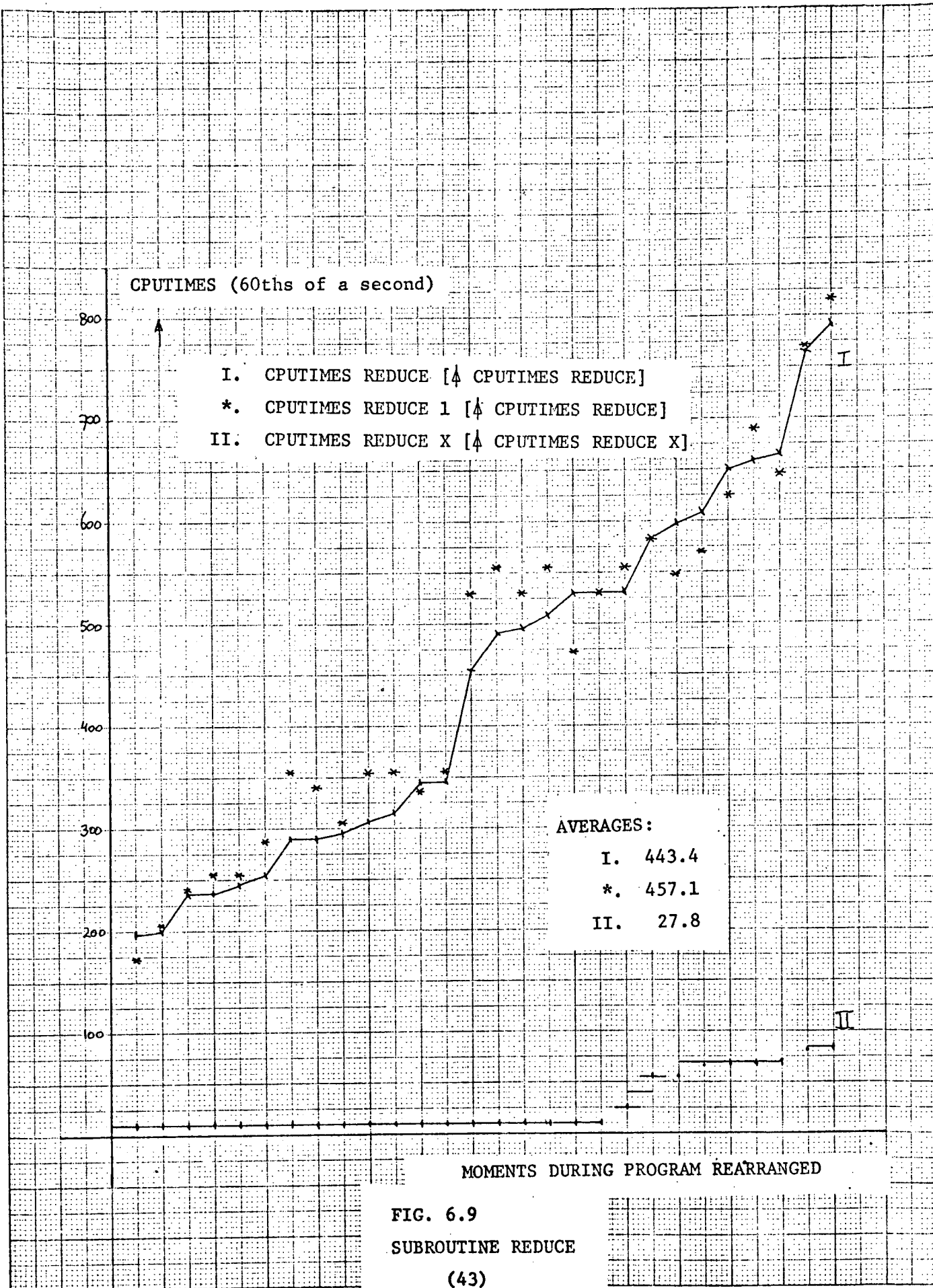
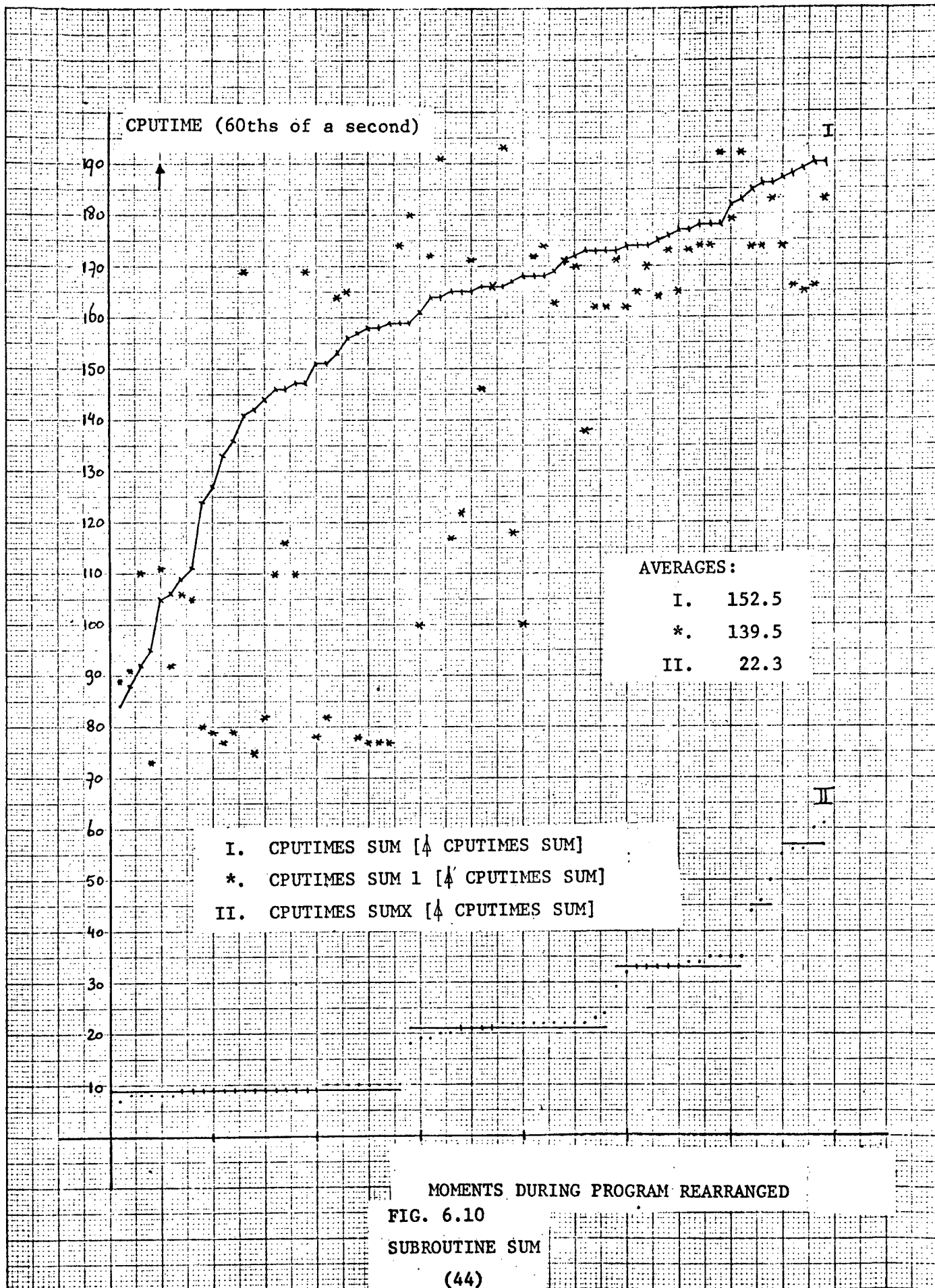


FIG. 6.9
SUBROUTINE REDUCE
(43)



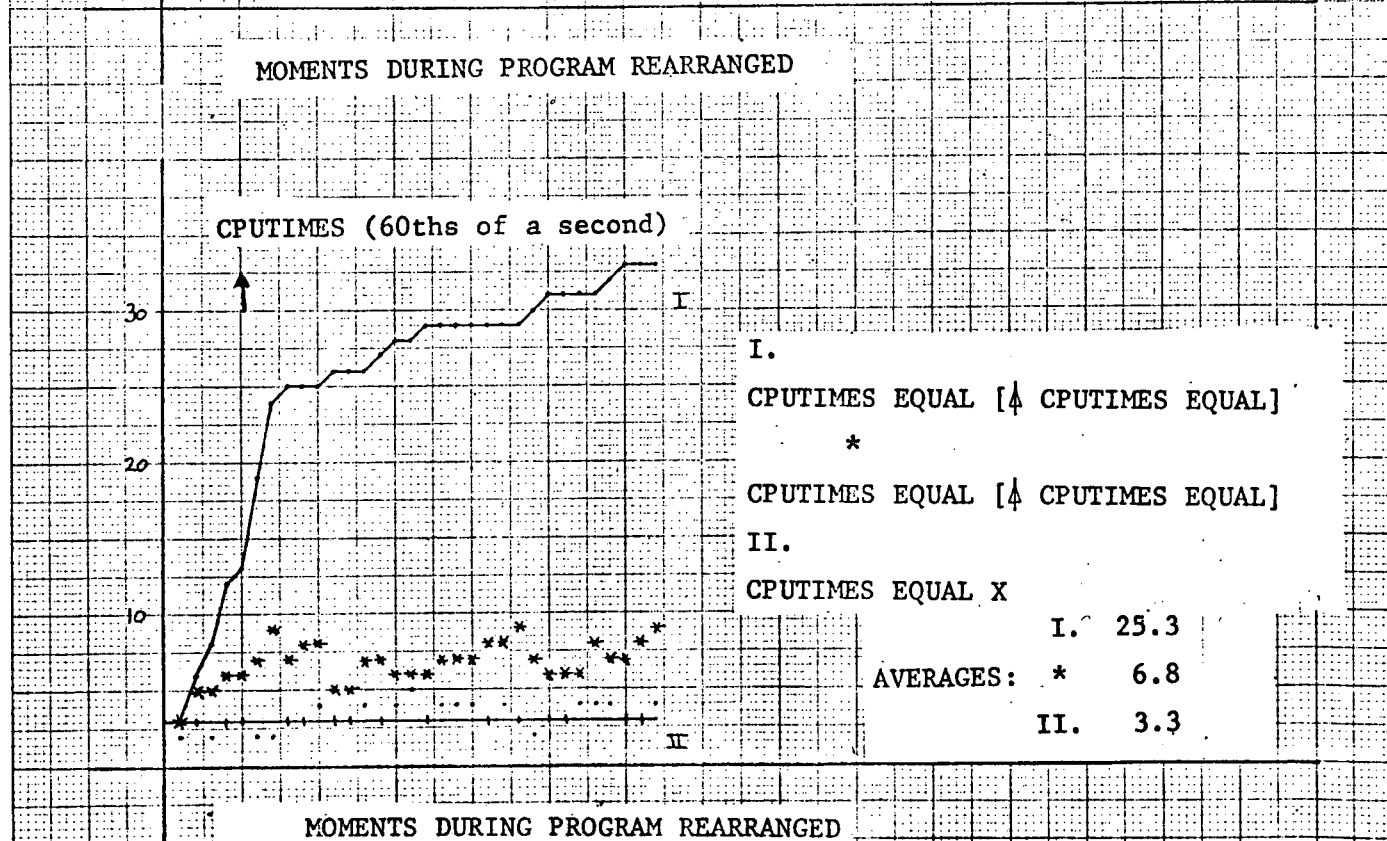
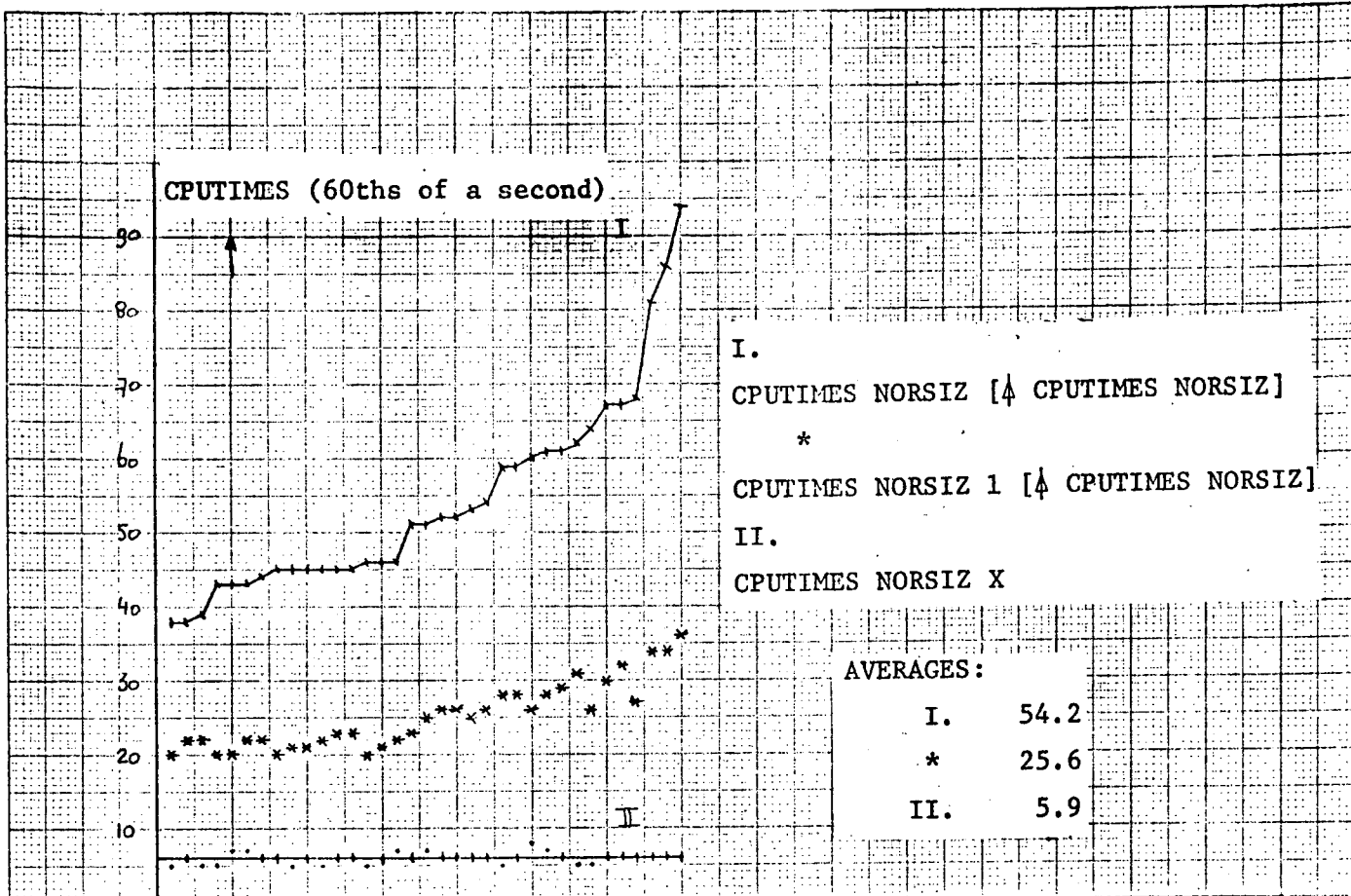


FIG. 6.11

SUBROUTINES NORISZ & EQUAL

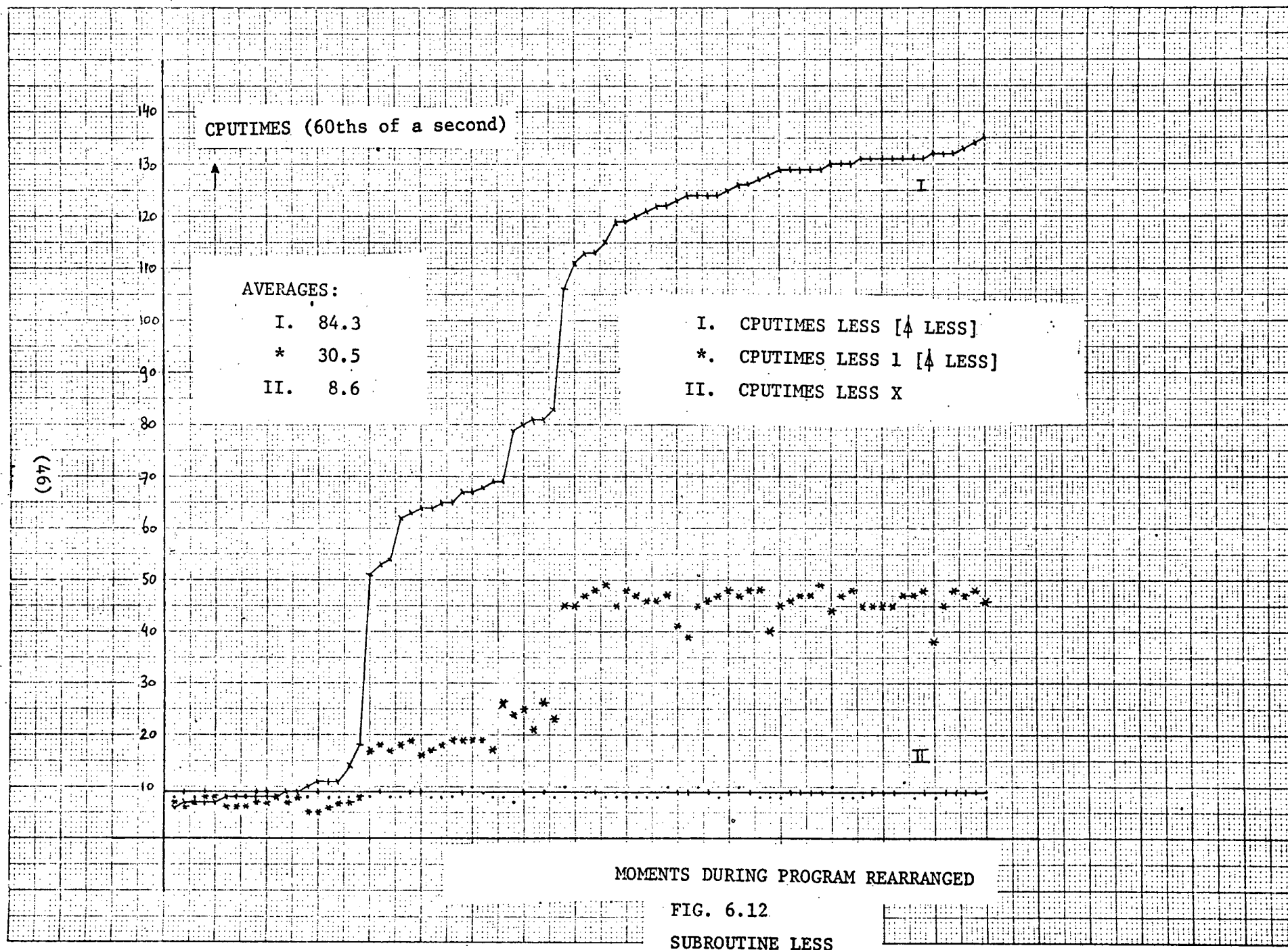


FIG. 6.12

SUBROUTINE LESS

--R.C. Singleton's formulation of the FFT, Algorithm 338 of the Collected Algorithms of the CACM [15], from ALGOL to FORTRAN. This algorithm is based on Singleton's approach to implementing the original Cooley-Tukey algorithm and the background material is contained in [14].

The six procedures, COMPLEXTRANSFORM, REALTRANSFORM, FFT2, REVFFT2, REORDER and REALTRAN were coded in addition to a main program which was used to read the input from data cards and then call COMPLEXTRANSFORM or REALTRANSFORM as appropriate. Following the observation made earlier in this report regarding coding in FORTRAN, double precision arithmetic was used throughout. For the actual tests to be described shortly only the MAIN program and the subroutines corresponding to the procedures for COMPLEXTRANSFORM, FFT2 and REORDER were compiled and used.

The equivalent *APL* function *FFT* was a modified version of an algorithm given by A.L. Jones, IBM, Endicott, N.Y. in the *APL Quote-Quad* [16]. The algorithm was modified to exploit improvements in *APL*\360, the IBM Program Product since Jones' algorithm was distributed. (He has since distributed an improved version.) The variant also provided both forward and inverse transforms and a scaling in both directions.

In each of the cases run, for both FORTRAN and *APL*, the forward transform and the inverse were calculated, invoking two calls to COMPLEX TRANSFORM. This was done to provide a check in returning to the original data. In the cited results both the FORTRAN and *APL* results agree to 10 significant places.

The use of FFT, in both environments, requires $2*N$ data for some N . Due to storage of temporaries and calculating with reals (long or 8 byte representation for floating point operations), *APL* is restricted to those cases where $N \leq 8$ for workspace sizes of 36 K. This limitation comes from the dynamic data size and while the restriction of size is much less than the number of

points normally used for the FFT, where N is usually in the range of 12 or so, the time and space trade offs may be seen.

7.1 Tests and Results for the FFT

The tests used data of the form

$$\nabla Z \leftarrow CRT N;T$$

$$[1] \quad Z \leftarrow ((2*N) \rho (T \rho 1), (T+2*N-1) \rho 0), [0.5] 0$$

$$\nabla$$

In general, the actual data has no real significance for the tests at hand; rather, the size of the problem is governed by N because $(2*N) = 1 \uparrow \rho CRT N$. The significance of the FFT is that the time, or number of calculations is proportional to $N \times 2*N$ for $2*N$ data points. The time to execute $\sim 1 FFT \sim 1 FFT CRT N$, or its FORTRAN equivalent, for $N = 16$ is summarized by:

(times in 60th's of a second)			
N	APL	FORTRAN	
		Compile, Load, Go	EXECUTION
1	39.4	3154	36
2	62.6	3143	42
3	96.4	3172	54
4	149.2	3190	68
5	255.4	3211	88
6	489.6	3268	141

The sizes are (in bytes):

APL	FORTRAN	
		LOAD MODULE
528	4752*	30,584**

* Includes 484 bytes of MAIN program to read input.

** Includes 4124 bytes of static COMMON to pass data to subroutines and 12 FORTRAN subroutines, such as IHCEFIOS* for I/O (21,708 bytes).

In FORTRAN the average time for all runs, for compilation, Link editing, and scheduling was 2466.3 60ths of a second. Comparing only the size of the programs the ratio is 528 bytes for *APL* to 4502 bytes for FORTRAN or 1 to 8.5. When the *APL* function is compared to the FORTRAN load module, the ratio is then 528 to 30,336 or 1 to 57. Carrying this comparison to one of total space we must compare the work space size, 36 K, to that needed for compile, load and execution, 160 K, giving a ratio of 1 to 4.44. If we take into the calculation the size of the interpreter, then (if a 1 workspace system would be a possibility) the ratios would be 124 K (88+36) to 160 K or 1 to 1.21. Placing relevance on any one of these ratios, (or other suggested comparisons, for that matter), is not a straightforward task. Comparing the direct program sizes does not measure the space dynamically allocated for data and for temporaries created during execution of the *APL* function. At the same time, part of the FORTRAN code is contained in the run time package, and yet an attempt to compare the *APL* function size to the FORTRAN program with the run time package overlooks the fact that *APL*'s structure requires the workspace and a great deal of an *APL* function's support is in the interpreter. Including the size of the interpreter in the calculation does not take into account the fact that the interpreter may be shared whereas run time packages generally are not. On the other side of the coin, the space used in the compile/execute cycle may be overlayed whereas the interpretive execution requires more nearly complete residency when attempting to use *APL* in a batch fashion.

The times of execution for the FFT would be expected to grow with $N \times 2 \times N$ for $2 \times N$ points, and the FORTRAN times when plotted on a semi-log scale have an almost linear relationship with N . The *APL* times are somewhat slower and show a growth greater than linear and approaching quadratic when plotted on the

same scale as the FORTRAN data. It is interesting to note that nowhere are the *APL* execution times comparable with the FORTRAN execution times, but over most of the range of *N* considered here the *APL* times are less than the FORTRAN scheduler times.

8.0 A NASA APPLICATION PROGRAM

In order to get some measure of utility in the application of interpretive techniques it was imperative to study one or more application programs typical of those encountered by scientists and engineers at Goddard Space Flight Center. The program supplied us by NASA Goddard was one written by M. Javid [17] when he was a visiting scientist at Goddard. The program, hereafter called the NASA Radiation Pattern Program, takes the geometry of a dish antenna, excited by an arbitrary primary feed, and calculates the resulting field at specified angular increments for Theta and Phi in a spherical coordinate system.

This particular program is of interest because in addition to being typical of the work of scientists and engineers, Javid developed the radiation pattern in *APL* and then from that a FORTRAN version was programmed for actually running the program. The effective use of *APL* in this fashion is reported by Javid in The Use of *APL* at Goddard Space Flight Center (C.J. Creveling Ed.) [18]. This type of use of *APL* only partially relates to the third category of use of *APL* which has been mentioned on page 2 of this report. Even though no compiler currently exists for *APL*, success has been found by using *APL* for algorithmic development with subsequent reprogramming in another language; see Kolsky [19] for another instance of this technique.

We were provided with a Xerox copy of a listing of the FORTRAN program along with the report [17], a Xerox copy of the *APL* functions, and the collection of papers edited by Creveling [18]. From this collection of material inferences

about this kind of program were to be drawn.

8.1 Program Characteristics and Programming Problems

The first task was to get Javid's FORTRAN H program running at Syracuse University. Unfortunately, a running program deck was not available and the quality of reproduction of the copy was lacking due to either lack of contrast or break-up in reproduction of the characters. Much time, both by man and computer, was spent removing errors of punching and program misinterpretation. Eventually success was achieved for the FORTRAN program and the availability of the original *APL* version and the descriptive material were invaluable in accomplishing this.

The program may be characterized by having a small amount of input data: the number of increments for Theta and Phi; the diameter of the reflector, which has rotational symmetry; the focal length; and the wave length. The nature of the geometry, and that of the primary feed, is implicit in the program. The *APL* function coded by Javid deals only with parabolic antennas, and we restricted ourselves to duplicating these cases.

It must be noted that if the flexibility is achieved by alternate coding, then additional effort in tailoring the program to the requirements of the problem must be made on a case by case basis.

The intermediate calculations are performed in a Cartesian coordinate system rather than one of spherical coordinates. In order to calculate the field at an arbitrary point, the circular antenna is divided into annular rings, the number of which is a function of the dish size and the wavelength. Each ring is divided into a number of segments such that each segment has approximately the same area as any other segment in other rings. An approximation of the field contribution of each segment is computed and then all of the contributions of the

segments are summed to provide, by superposition, an approximation, to the limiting case of arbitrarily small segments, of the surface integral.

The field is calculated at each of the (Number of Theta increments) \times (Number of Phi increments) points by a doubly nested looping procedure. After normalization there is a translation from cartesian coordinates to a spherical system to give the radiation pattern.

8.2 Recasting The Original APL Program

Javid's original collection of functions were written at a time before the circular functions were added as APL primitives. Thus, an obvious step was to delete the APL code for the functions *SIN X* and *COSX* use *10X* and *20X* respectively in the body. This minor change is reflected in Figure 8.1.

Lines 125 and 128 of *BEAM* have an error in them. Lines 124 to 129 are used to translate from cartesian to spherical coordinates and for both the real and imaginary components in the Theta direction

$$I_{\theta}^{r, i} = \cos \theta \cos \emptyset I_x^{r, i} + \cos \theta \sin \emptyset I_y^{r, i} - \sin \theta I_z^{r, i}$$

and not

$$I_{\theta}^{ri} = \cos \theta \cos \emptyset I_x^{ri} + \cos \theta \cos \emptyset I_y^{r, i} - \sin \theta I_z^{r, i}$$

as shown in lines 125 and 128. Even with the corrections an examination of the ancillary functions

HXR, *HXI*, *HYR*, *HYI*, *HZR*, *HZI*, which are used to calculate the Real and Imaginary components of the source field, *H*, the *X*, *Y*, and *Z* directions based on the *x*, *y*, and *z* values (of course these depend on *r*, θ , and \emptyset) points to other changes. These functions have a large dependence upon the use of global variables with little use (in *HXR*, *HXI*, *HZR*, *HZI*) of the arguments, and this and other considerations suggest treating a

```

VBEAM[ ]V
V RES←AR1 BEAM AR2
[1] DIA←30
[2] LMDA←0.425
[3] DFID←3
[4] DTAD←3
[5] TPI←2×3.141592654
[6] DR←TPI÷360
[7] DFI←DFID×DR
[8] DTA←DTAD×DR
[9] K←TPI÷LMDA
[10] DRHC←S+LMDA÷4.731666667
[11] R←[(DIA÷2)÷S
[12] M←999
[13] RI←(R+2)ρ0
[14] NSUM←(R+2)ρ0
[15] J←0
[16] I←0
[17] B13:I←I+1
[18] SUM←0
[19] B12:J←J+1
[20] →(J>R)ρB10
[21] DSUM←6×J
[22] →(DSUM>M)ρB14
[23] SUM←SUM+DSUM
[24] →(SUM>M)ρB11
[25] →B12
[26] B11:RI[I+1]←J-1
[27] NSUM[I+1]←SUM-DSUM
[28] J←J-1
[29] →B13
[30] B14:'STORAGE IS INSUFFICIENT FOR THE FOLLOWING RING:'
[31] J
[32] RI[I+1]←J-1
[23] NSUM[I+1]←SUM
[34] I←I+1
[35] →B15
[36] B10:'END OF REFLECTOR SPECIFICATION'
[37] RI[I+1]←J-1
[38] NSUM[I+1]←SUM
[39] I←I+1
[40] B15:LIR←I
[41] 'TOTAL NUMBER OF ELEMENTARY AREA CONTRIBUTING TO THIS COMPUTATI-
ON IS:'
[42] +SUMDS←+/NSUM
[43] NTX←(9,((AR1+1)×AR2+1),LIR-1)ρ0
[44] VX←Mp1
[45] VY←Mp1
[46] VZ←Mp1
[47] VNX←Mp1
[48] VNY←Mp1
[49] VNZ←Mp1
[50] VDS←Mp1
[51] VHXR←Mp1
[52] VHXI←Mp1
[53] VHXR←Mp1
[54] VHXI←Mp1
[55] VHXR←Mp1
[56] VHXI←Mp1
[57] IR←0
[58] B16:IR←IR+1

```

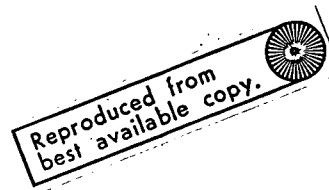


FIG 8.1
BEAM (ORIGINAL)

```

[121] IZR+IZR+((AZR*CKD)-AZI*SKD)*VDS[I]
[122] IZI+IZI+((AZR*SKD)+AZI*CKD)*VDS[I]
[123] →B7
[124] B6:CRR+(STA*CFI*IXR)+(STA*SFI*IYR)+CTA*IZR
[125] CTR+(CTA*CFI*IXR)+(CTA*CFI*IYR)-STA*IZR
[126] CFR+(-SFI*IXR)(CFI*IYR) → see p. 52
[127] CRI+(STA*CFI*IXI)+(STA*SFI*IYI)+CTA*IZI
[128] CTI+(CTA*CFI*IXI)+(CTA*CFI*IYI)-STA*IZI
[129] QFI+(-SFI*IXI)+CFI*IYI → see p. 52
[130] AR2V+(CRR*2)+CRI*2
[131] AT2V+(CTR*2)+CTI*2
[132] AF2V+(CFR*2)+QFI*2
[133] MFX[;FPNC;IR]+CRR,CTR,CFR,CRI,CTI,CFI,AR2V,AT2V,AF2V
[134] AT2V
[135] →B9
[136] B4:→B16
      VSIN[ ]V
      V AN+SIN T
[1]  AN+10T
      V
      V COS[ ]V
      V AN+COS T
[1]  AN+20T
      V
      VDS[ ]V
      V DSI+X DS Y
[1]  DSI+TPI*(((RHO+0.5*S)*2)-(RHO-0.5*S)*2):2*NUM
      V
      VHYI[ ]V
      V HYII+X HYI Y
[1]  HYII+ZR2*SKR
      V
      VZII[ ]V
      V Z+X ZI Y
[1]  F+36
[2]  RHO2+(X*2)+Y*2
[3]  Z+(RHO2÷(4*X))-F
      V
      VHXRI[ ]V
      V HXRI+X HXR Y
[1]  HXRI+0
      V
      VHZRI[ ]V
      V HZRI+X HZR Y
[1]  YR2+Y÷R2
[2]  HZRI+YR2*CKR
      V
      VHZII[ ]V
      V HZII+X HZI Y
[1]  HZII+YR2*SKR
      V
      VHYRI[ ]V
      V HYRI+X HYR Y
[1]  R2+(X*2)+(Y*2)+Z*2
[2]  R1+R2*0.5
[3]  KR+K×R1
[4]  SKR+SIN KR
[5]  CKR+COS KR
[6]  SR2+Z÷R2
[7]  HYRI+SR2*CKR
      V
      VNXZ[ ]V
      V VNXZ+X NXZ Y
[1]  R1+((X*2)+Y*2)+Z*2)*0.5
[2]  VNXZ+X÷(Z-R1)
      V
      VNYZ[ ]V
      V VNYZ+X NYZ Y
[1]  R1+((X*2)+(Y*2)+Z*2)*0.5
[2]  VNYZ+Y÷(Z-R1)
      V
      VNZZ[ ]V
      V VNZZ+X NZZ Y
[1]  VNZZ+-1
      V

```

Reproduced from
best available copy.

FIG 8.1 (continued)

BEAM (ORIGINAL)

```

[59] →(IR=LIR)ρ0
[60] FPNO←0
[61] T←NSUM[IR+1]
[62] IEND←0
[63] I←1
[64] J←-1
[65] B1:J←J+1
[66] →(J=(RI[IR+1]-RI[IR]))ρB3
[67] JH←RI[IR]+J+0.5
[68] RHO←S×JH
[69] I←I-1
[70] IENDO←IEND
[71] NUM←6×JH+0.5
[72] DPHI←TPI:NUM
[73] IEND←IENDO+NUM
[74] B2:I←I+1
[75] →(I=IEND+1)ρB1
[76] IH←(I-IENDO)-0.5
[77] PHI←DPHI×IH
[78] VX[I]←X+RHO×COS PHI
[79] VY←Y+RHO×SIN PHI
[80] VZ[I]←Z+X ZI Y
[81] VNX[I]←X NXZ Y
[82] VNY[I]←X NYZ Y
[83] VDS[I]←X DS Y
[84] VNXR[I]←X HXR Y
[85] VHXI[I]←X HXI Y
[86] VHXR[I]←X HYR Y
[87] VHYI[I]←X HYI Y
[88] VHZR[I]←X HZR Y
[89] VHZI[I]←X HZI Y
[90] →B2
[91] B3:LI←-1
[92] B5:NI←NI+1
[93] →(NI=AR1+1)ρB16
[94] NJ←-1
[95] B9:NJ←NJ+1
[96] →(NJ=AR2+1)ρB5
[97] FPNO←FPNO+1
[98] IXR←IXI+IYR+IYI+IZR+IZI+0
[99] TA←NI×DTA
[100] FI←NJ×DFI
[101] STA←SIN TA
[102] CTA←COS TA
[103] CFI←COS FI
[104] SFI←SIN FI
[105] I←0
[106] B7:I←I+1
[107] →(I=T+1)ρB6
[108] AD←K×((VX[I]×CFI×STA)+VY[I]×SFI×STA) VZ[I]×CTA
[109] CKD←COS KD
[110] SKD←SIN KD
[111] AXR←(VNY[I]×VHZR[I])-VHYR[I]
[112] AXI←(VNY[I]×VHZI[I])-VHYI[I]
[113] IXR←IXR+((AXR×CKD)-AXI×SKD)×VDS[I]
[114] IXI←IXI+((AXR×CKD)+AXI×SKD)×VDS[I]
[115] AYR←VNXR[I]-VNX[I]×VHZR[I]
[116] AYI←VHXI[I]-VNX[I]×VHZI[I]
[117] IYR←IYR+((AYR×CKD)-AYI×SKD)×VDS[I]
[118] IYI←IYI+((AYR×CKD)+AYI×SKD) VDS[I]
[119] AZR←(VNX[I]×VHYR[I])-VNY[I]×VHXR[I]
[120] AZI←(VNX[I]×VHYI[I])-VNY[I]×VHXI[I]

```



FIG 8.1 (continued)

BEAM (ORIGINAL)

point as a 3 element vector and H as say a 2 by 3 matrix. This in turn offers a general reorganization of $BEAM$ along lines encountered in Section 6 of this report. The strategy would be to create an array which encompasses each of the ρ $THETA$ by ρ PHI points in both real and imaginary components in each of the x , y , and z directions. These values are then calculated for each of the segments found in all of the annular rings. If this number is N , then the array would be of a size which is

$$(\rho \text{ } THETA), (\rho \text{ } PHI), 2 \text{ } 3, N$$

A plus reduction along the last dimension approximates the integral and produces the answer in a cartesian coordinate system.

One immediate problem is that for $THETA$ and PHI increments of 3 degrees to cover say 90° in each of $THETA$ and PHI requires $30 \times 30 \times 2 \times 3 = 5400$ values, each using 8 bytes for storage and thus requiring 43200 bytes for the result. Intermediate calculations become even more demanding. The total number of segments contributing to the calculating is given by $N \leftarrow +/6 \times 1 \text{ } R$ where R is the number of rings.

$$+/6 \times 1 \text{ } R \leftarrow 6 \times +/1 \text{ } R \leftarrow 6 \times .5 \times R \times R + 1 \leftarrow 3 \times R \times R + 1.$$

R is dependent on the geometry and wave length; for say a 30 foot diameter antenna with a wave length of .425, R will be 167 and this means that N will be 84,168 as calculated by Javid's original APL antenna radiation program. This clearly indicates that 673,344 bytes would be needed to store these H values. Clearly, looping of some kind is imperative. The choice was to attempt to maintain all points for $THETA$ and PHI in three dimensions and two components of the complex numbers and then generate as many segments as space will allow.

The functions for doing this but neither reconverting to spherical coordinates nor computing the power (See lines 124-132 of Figure 8.1) are shown in Figure 8.2.

B

```

      VBEAM[ ]V
    V BEAM
[1]  INITIALIZE
[2]  S+LMDA÷4.731666666666667
[3]  L+1R+10.5×DIA+S
[4]  LOOP:→ON×1NUM<ρRHO
[5]  GETMORE
[6]  ON:→CONVERT×10=ρRHO
[7]  SI+NUM\ρRHO
[8]  XYZ←(SI+RHO) POINT SI+PHI
[9]  RHO←SI+RHO
[10] PHI←SI+PHI
[11] VNH←H XYZ
[12] KD←K×ANG+.×XYZ
[13] KD← 2 1 0.0 3 1 2 4 Q(3,ρKD)ρKD
[14] NV←N XYZ
[15] VNH←(NV CROSS VNH[1;;]),[0.5] NV CROSS VNH[2;;]
[16] VNH← 2 3 1 4 5 Q(AR,ρVNH)ρVNH
[17] VNH←(-÷VNH×KD),[0.5]÷VNH×KD
[18] KD←(2,AR,3,SI)ρSI+DS
[19] I←I++/VNH×KD
[20] DS←SI+DS
[21] →LOOP
[22] CONVERT:'ADD THE CTS CODE HERE'
    V

      VINITIALIZE[ ]V
    V INITIALIZE
[1]  'ENTER NUMBER OF STEPS (OF 3 DEGREES) FOR THETA AND P
    HI'
[2]  AR←1+[]
[3]  'REFLECTOR DIAMETER='
[4]  DIA←[]
[5]  'FOCAL LENGTH ='
[6]  F←[]
[7]  'WAVELENGTH ='
[8]  K←02+LMDA+[]
[9]  I←(2,AR,3)ρ0
[10] DS←PHI+RHO+10
[11] TA←0(3×1+11+AR)+180
[12] FI←0(3×1+11+AR)+180
[13] ANG← 3 1 2 Q(1 1 2 0.0Q(φAR)ρTA)×(2 1 0.0ARρFI),[1] 1
    V

      VGETMORE[ ]V
    V GETMORE;J;NO;ΔPHI;ΔRHO
[1]  BLD:→0×11>ρL
[2]  J←1+L
[3]  L←1+L
[4]  ΔPHI←02+NO+6×J
[5]  PHI←PHI,ΔPHI×-0.5+1NO
[6]  RHO←RHO,ΔRHO←NOρS×J-0.5
[7]  DS←DS,S×ΔRHO×ΔPHI
[8]  →BLD×1NUM≥ρRHO
    V

```

FIG 8.2
BEAM (MODIFIED)

```

      VPOINT[[]]V
      V XYZ←RHO POINT PHI
[1] XYZ←(3,ρRHO)ρ(RHO×2φPHI),(RHO×1φPHI),(RHO÷4×F)-F
      V

      VH[[]]V
      V Z←H XYZ;R;MR;SR;T
[1] Z←(θ 1 0 +XYZ)+(2,SR←ρR)ρMR←(R←+XYZ*2)*
      0.5
[2] T← 2 1 0.0K×MR
[3] Z←(2 3 ,SR)ρ(SRρ0),(Z[1;]×-T[1;]),(Z[2;]×T[1;]),(SRρ0
      ),(Z[1;]×T[2;]),Z[2;]×-T[2;]
      V

      VH[[]]V
      V VN←N XYZ
[1] VH←((-1 0 +XYZ)+(-1 0 +ρXYZ)ρ(, 2 0 +XYZ)-(+XYZ*
      2)*0.5),[1] 1
      V

      VCROSS[[]]V
      V Z←A CROSS B
[1] Z←-((1θA),[0.5] 2θA)×(2θB),[0.5] 1θB
      V

      VCTS[[]]V
      V Z←L CTS R;M
[1] R←(Mφ[2] 2 2 3 ρ 1 1 1 1 1 -1)×((M← 2 3 ρ 0 0 0 1 1 1
      )φ[2] 2 2 3 ρ 1 1 2 2 2 1)OR← 2 3 1 φ 2 3 2 ρR
[2] Z←((R[1;;],[1] 1 1 0)×φR[2;;],[1] 1 1 0)+.×L
      V

```

A CTS GIVES THE CARTESIAN TO SPHERICAL CONVERSION FOR A SINGL
 A TO USE IT REQUIRES CONDITIONING THE ARRAY RESULTING FROM BE

FIG 8.2 (continued)
 BEAM (MODIFIED)



8.3 Size of Computations and Their Implications

In order to check the revised *APL* program against FORTRAN the modified *APL* program was compared against the FORTRAN H version. The original data given by Javid in [18] was miniaturized by selecting a similar number of *THETA* by *PHI* increments and not changing the wave length. The radius and the focal length, however, were reduced by a factor of 10 (from 30 and 36 feet to 3.0 and 3.6 feet respectively). This decreases the area and hence the computations by 2 orders of magnitude. The answers would not be numerically accurate for such a problem but the amount of computation would be. The computations were done so as to produce results in a cartesian coordinate system to check whether the two programming efforts produced equivalent results up to that point.

The results of the first test may be summarized by:

SYSTEM	Time		Program Size (bytes)
	Compile Load and Go (sec÷60)	Go (sec÷60)	
<i>APL</i>	-	25,681 (7 min, 8sec 1,60th)	2980 (125K workspace)
FORTRAN	7656 (2 min, 7 sec., 36 60th)	2059 (34 sec 19 60th)	18,886 (Program) 99,328 (load module) (includes 57,552 bytes of COMMON and 22,894 bytes of subroutine for I/O etc.)

This makes the execute step in FORTRAN 12.47 times as fast as the *APL* execution, with the *APL* program 6.34 times as compact as the FORTRAN program. Taking into account 160 K partitions for compiling and about 100 K bytes needed at execute time compared with using 125 K workspaces in *APL*, *APL* is 2.91 times as

costly as FORTRAN in this case when measured in terms of core residency times (byte-seconds), a simple product of space and time.

If we expect the time of execution on the actual program to be increased by a factor of 100 due to increasing the diameter and focal length by a factor of 10, then one could expect a CPU execute time of 11 hours, 53 minutes and 22 seconds in *APL*.

This time was too excessive to permit full execution within the scope of this work; however, due to the way in which the area of the dish is divided we may time a portion of the program and estimate with reasonable accuracy the time involved.

Since the full test was made with 16 increments for *THETA* and 1 for *PHI* while the "mini" antenna test had 6 increments for *THETA* and 3 for *PHI*, some compensation for the estimated times would have to be made to compare the two figures for the actual test.

Based on 83.75 minutes for CPU time (12.9% of the work) the *APL* version of the radiation pattern program would run for 10 hours and 49 minutes.

The FORTRAN H program running for 20.57 minutes and accomplishing 44% of the work has an estimated time of 46.8 minutes. This leads to a ratio of 13.87.

When we adjust the amount of *THETA* and *PHI* points for which the calculations are done for the "mini" test as opposed to and the full scale antenna, the *APL* estimates are consistent with the change in the amount of work in going from the "mini" antenna to the full scale problem, two orders of magnitude.

Some observations may be drawn from the above. First, problems of this size are reasonably large, even in a conventional sense for a system 360 model 50; the times projected for an interpretive execution in *APL* place that mode of solution beyond practicality. Moreover, the problem is of such a nature that attempts to trade space for execution speed by removing loops

lead to difficulties in size.

The present implementation of *APL* requires the workspace size to hold all temporary results and removal of explicit looping by using an array approach for computation implies large (in this case very large) temporary results. The fact that the algorithm for this problem can be written so as to have essentially no loops is of value only if the time and space requirements of the implementation allow the exploitation of such a formulation. Unfortunately, this is not the case at present. Large increases in workspace size or in physical space for temporaries negates the favorable code density of *APL*.

An *APL* implemented on a machine having virtual memory would allow for problems of this sort, the availability of large conceptual arrays while keeping the working set of physical items within reason. Of course the same system could be applied to the FORTRAN program, but its use of explicit looping in the algorithm has less requirement for such automatic paging to manage the data.

The ability of *APL* to trade time for space is thus, in this case, somewhat a function of the implementation. A change in implementation strategy might reduce the cost of interpretation, even without a virtual machine. Such a change would probably not change the overall results, but allowing a greater degree of looping in the same amount of computer time would permit a reduction in space requirements. This could make *APL* more attractive if the original consideration had been one of sacrificing cpu cycles to gain space.

The value of *APL* to specify and develop algorithms for implementation in other languages is well established by this example.

In fact the time to get the new *APL* version running was less than that to keypunch and debug the FORTRAN version using its listing.

9.0 CONCLUSIONS

This study has examined a number of areas of programming related to scientific problems. These range from the very large- where the total number of values for temporaries and final results in a typical problem could run into billions of bytes of storage, down to the small where both the source code and the generated data are in the range of hundreds of bytes or less.

We have been concerned in this range of tasks with the use of an interpretively based language, *APL*, in comparison with compiled code, as generated by FORTRAN. While a number of problem areas examined have been implemented for both batch and a time sharing environment, we were primarily concerned with execution times which give emphasis to the more traditional batch mode of operation. In that mode of operation much of the compilation may indeed be recompilation and in general little is said of the time and hence the cost of scheduling, compiling and link editing.

The studies here did not address the issue of the efficiency of programming in *APL* as opposed to more traditional languages. Such a study, if objective, would be valuable, but usually studies comparing an interactive approach versus batch programming, even in the same language, often find a greater variation among programmers than between methodologies.

Rather, these examples have been pointed toward issues of: 1) timings for both execution and in the FORTRAN environment, total time for compilation, loading and execution and 2) space requirements. Toward these ends FORTRAN H OPT = 2 was used as the compiler, and in both the FORTRAN and *APL* cases the system was run without confluence.

Breed and Lathwell [20] have previously reported execution times for *APL* which are five to ten times slower than compiled code. We have not found results which uniformly contradict that

range of results. There are cases reported herein where compiled code is from 4 to 15 times as fast as *APL* with the larger ratios occurring for very large problems.

There are also cases where *APL* runs faster than compiled FORTRAN measured at the execute step. These instances tend to be those such as inner products and *DOMINO* and others where reasonably sophisticated FORTRAN programs are themselves replaced by an *APL* primitive.

There are a number of instances where FORTRAN in the Go step was faster than *APL* but compared to Compile, Load and Go, *APL* has the advantage. Thus, if there is even reasonable need to recompile during development, *APL* has a cost advantage over the entire range of use.

APL code is in the order of 10 times as dense as compiled FORTRAN. The figures do not include data space in *APL* in that it is dynamically allocated but the figures do include the pre-dimensional space allocated in FORTRAN. Thus, in the present implementation, when *APL* is written to take advantage of the array capabilities of the language, then the space requirements for *APL* will increase greatly. Of course that space is upper bounded by the workspace size but the code density takes an additional meaning in any system where the computer hardware performs a mapping process in memory hierarchy independent of software. This could be significant in virtual or cache memory systems.

The size of the *APL* interpreter is fairly large, 88 K bytes in *APL\360* , but the run time support packages for FORTRAN programs are often about 1/4 of that size and in general are not shared among processes. Thus, if multiprogramming is done, after four or five FORTRAN programs are executing the size of *APL* interpreter has probably been used to support the running programs anyway.

In general for small problems, those that fit well within the defacto standard 36 K workspaces, *APL* compares very favorably with compiled code, taking into account both time and space. An improvement of a factor of 3 or 4 would make *APL* extremely competitive over much of the range of situations encountered in this report. Improving the speed of *APL* by 50 to 100 per cent is no doubt obtainable without a major reimplementatation effort.

Two observations are worth noting as closing remarks.

First, to be at all competitive, algorithms must be written in "good" *APL* which often means rethinking the problem, but even with that in mind *APL* may be competitive not because it and the algorithms being executed are well written, but rather because the batch processing is less efficient than we have been willing to admit.

Second, the present version of *APL* 360 is not radically changed from the original implementation which was an experimental research tool, implemented to provide reliable support of terminals running problems somewhat more restricted than those encountered in normal batch processing. The accumulated and published knowledge concerning efficient implementation of *APL* is, at this writing, pretty scant. There is not yet a broad base of experience founded on actually trying different implementation strategies which have been targeted at open competition with traditional processing methods.

While this study does not establish *APL* to be as effective as we would like it to be, it is no doubt better than many thought it to be. We may anticipate research and development to improve it, beyond what we now have. In its use it is certainly superior in many areas and use will probably confirm its effectiveness in a broader sense, but in the interim we must agree with Frank Plumpton Ramsey that, "We are in the ordinary position of scientists of having to be content with piecemeal improvements; we can make several things clearer, but we can not make anything clear."

References

- [1] Iverson, K.E., A Programming Language, (1962) John Wiley and sons, New York.
- [2] Falkoff, A.D., K.E. Iverson and E.H. Sussenquth, "A Formal Description of System/360," IBM Systems Journal, 3,3 (1964), pp. 198-262.
- [3] Falkoff, A.D. and K.E. Iverson, APL\360 Users Manual, (1968) IBM Corporation.
- [4] Gilman, L. and A.J. Rose, APL\360: An Interactive Approach, (1970) John Wiley and Sons, New York.
- [5] APL\360 - OS/DOS General Information Manual, IBM Corp. (GM20-0850).
- [6] Jenkins, M.A., "The Solution of Linear Systems of Equations and Linear Least Squares Problems in APL," Technical Report No. 320-2989, June 1970, IBM, New York Scientific Center.
- [7] Jenkins, M.A., "Domino - An APL Primitive Function for Matrix Inversion - Its Implementation and Applications," Proceedings SHARE XXXVII, Vol. 1, pp. 380-388.
- [8] Westlake, J.R., "A Handbook of Numerical Inversion and Solution of Linear Equations," John Wiley and Sons, Inc., New York, 1968.
- [9] Hellerman, H., Digital Computer Systems Principles, McGraw-Hill, New York, 1967.
- [10] Piatkowski, Computer Programs Dealing with Finite State Machines Part II, Department of Electrical Engineering, University of Michigan, Ann Arbor, Michigan, July 1967 (AD 658 001).
- [11] Foster, Garth H., "Using APL to Investigate Sequential Machines," Technical Applications Papers NEREM-70 (70 C 63), pp. 120-127.
- [12] Hartmanis, J. and R.E. Stearns, Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Englewood Cliffs, New Jersey, 1966.
- [13] Brigham, E.O. and R.E. Morrow, "The Fast Fourier Transform," IEEE Spectrum, December 1967.
- [14] Singleton, R.C., "On Computing the Fast Fourier Transform," Communications of the Association of Computing Machinery, 10, 10 (October 1967).

References Continued

- [15] Singleton, R.C., "Algorithm 338-The Fast Fourier Transform," Collected Algorithms of the Communications of the Association for Computing Machinery.
- [16] Jones, A.L., "FFT - A Fast Fourier Transform," The APL Quote-Quad.
- [17] Javid, M., A Digital Computer Program for Calculating the Radiating Pattern of an Antenna of Arbitrary Geometry with Arbitrary Primary Feed, Goddard Space Flight Center, Greenbelt, Maryland, Document X-200-67-639 (1967).
- [18] Experimental Use of A Programming Language (APL) at the Goddard Space Flight Center, (C.J. Creveling, Ed.) Goddard Space Flight Center, Greenbelt, Maryland, Document X-560-68-420, November 1968, pp. 9-14.
- [19] Kolsky, H., "Problem Formulation in APL," IBM System Journal, 8, 3(1969), pp. 204-219.
- [20] Breed, L.M. and R.H. Lathwell, "The Implementation of APL/360," Interactive Systems of Applied Mathematics, (Klerer and Reinfelds, Eds.) (1968), Academic Press, New York, pp. 390-399.

APPENDIX A

FAST FOURIER TRANSFORM
PROGRAMS (*APL* and FORTRAN)

(67)

C

▽FFT[[]]▽

```

▽ Z←I FFT X;J;K;L;M;N;O;S;W;A;Δ
[1] W←[2*-O-1K,0pS+1-2x~O+1pM+12pJ+1H+~1+ρX
[2] Z←X[;(K+O)+(Mp2)1L+Θ(Mp2)TJ-O]
[3] X← 2 1 °.○○((×I)×O-J)+~1+W
[4] Z←Z[;J-,W[K]×L[K;]]+(pZ)p(-X[;Δ]×Z[;A]),+X[;Δ+O+NpW[S+M-K]×
-O-12×W[K]]×ΘZ[;Δ+J+,W[K]×O=L[K;]]
[5] →((M+O)>K+K+1)/4
[6] Z←Z÷N*0.5
▽

```

```

//
//C95336 JOB
// (0643,EE,5,5),'FFT338',REGION=160K
//PRF EXEC FORTHCLG,PARM.FORT='SOURCE,MAP,OPT=2'
//FORT.SYSIN DD *
C MAIN PROGRAM TO COMPUTE CACM ALGORITHM 338
C ALGOL PROCEDURE FOR THE
C
C FAST FOURIER TRANSFORM
C
C BY RICHARD C. SINGLETON
C
C *****
C
C MAIN PROGRAM FOR INPUT AND OUTPUT FOR FFT
C USES PROCEDURES COMPLEXTRANSFORM AND REALTRANSFORM
C
COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,B
INTEGER*4 M,N,I,J,L
LOGICAL INVRSE
READ (5,1000) L,INVRSE
1000 FORMAT (2X,I8,2X,L1)
N=2**L
READ (5,1002) (A(I),I=1,N)
1002 FORMAT (4D20.10)
READ (5,1002) (B(I),I=1,N)
DO 1090 J=1,2
M=L
CALL CTRFRM
1090 INVRSE=.NOT. INVRSE
STOP 9999
END
SUBROUTINE CTRFRM
C
C PROCEDURE COMPLEXTRANSFORM (A,B,M,INVERSE)
C USES PROCEDURES FFT2,REORDER
C
COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,B,P,Q
INTEGER*4 M,N,J,NA,NAA
LOGICAL INVRSE
N=2**M
Q=1.000/DSQRT(DFLOAT(N))
P=Q
IF (.NOT. INVRSE) GO TO 10
Q=-Q
NA=IABS(N-1)+1.0000001
DO 9 NAA=1,NA
J=N-NAA
9 B(J+1)=-B(J+1)
10 CALL FFT2 (N)
CALL REORDR (N,.FALSE.)
NA=IABS(N-1)+1.0000001
DO 12 NAA=1,NA
J=N-NAA
A(J+1)=A(J+1)*P
12 B(J+1)=B(J+1)*Q
RETURN

```

```

END
SUBROUTINE FFT2 (KS)
C
C      PROCEDURE FFT2 (A,B,N,M,KS)
C      USES NO OTHER PROCEDURES
C
COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,B,A0,A1,A2,A3,B0,B1,B2,B3
REAL*8 RAD,C1,C2,C3,S1,S2,S3,CK,SK,SQ
INTEGER*4 M,N,KS,C(9),NA,NAA
INTEGER*4 K0,K1,K2,K3,SPAN,J,JJ,K,KB,KN,MM,MK
LOGICAL INVRSE
SQ=0.707106781187
SK=0.382683432366
CK=0.92387953251
C(M+1)=KS
MM=(M/2)*2
KN=0
NA=IABS(M-1)+1.0000001
DO 240 NAA=1,NA
K=M-NAA
240 C(K+1)=C(K+2)/2
RAD=6.28318530718/(C(1)*KS)
MK=M-5
C
C      LABEL 250 IS L IN ALGOL
C
250 KB=KN
KN=KN+KS
IF (MM .EQ. M) GO TO 260
K2=KN
K0=C(MM+1)+KB
C
C      LABEL 252 IS L2 IN ALGOL
C
252 K2=K2-1
K0=K0-1
A0=A(K2+1)
B0=B(K2+1)
A(K2+1)=A(K0+1)-A0
A(K0+1)=A(K0+1)+A0
B(K2+1)=B(K0+1)-B0
B(K0+1)=B(K0+1)+B0
IF (K0 .GT. KB) GO TO 252
260 C1=1.0
S1=0.0
JJ=0
K=MM-2
J=3
IF (K .GE. 0) GO TO 275
GO TO 294
C
C      LABEL 270 IS L3 IN ALGOL
C
270 IF(C(J+1) .GT. JJ) GO TO 272
JJ=JJ-C(J+1)
J=J-1
IF (C(J+1) .GT. JJ) GO TO 272
JJ=JJ-C(J+1)
J=J-1

```

```

      K=K+2
      GO TO 270
272  JJ=C(J+1)+JJ
      J=3
C
C      LABEL 275 IS L4 IN ALGOL
C
275  SPAN = C(K+1)
      IF (JJ .EQ. 0) GO TO 282
      C2=JJ*SPAN*RAD
      C1=DCOS(C2)
      S1=DSIN(C2)
C
C      LABEL 280 IS L5 IN ALGOL
C
280  C2=C1**2-S1**2
      S2=2.0*C1*S1
      C3=C2*C1-S2*S1
      S3=C2*S1+S2*C1
282  NA=IABS(SPAN-1)+1.0000001
      DO 290 NAA=1,NA
      K0=KB+SPAN-NAA
      K1=K0+SPAN
      K2=K1+SPAN
      K3=K2+SPAN
      A0=A(K0+1)
      B0=B(K0+1)
      IF(S1 .NE. 0) GO TO 284
      A1=A(K1+1)
      B1=B(K1+1)
      A2=A(K2+1)
      B2=B(K2+1)
      A3=A(K3+1)
      B3=B(K3+1)
      GO TO 286
284  A1=A(K1+1)*C1-B(K1+1)*S1
      B1=A(K1+1)*S1+B(K1+1)*C1
      A2=A(K2+1)*C2-B(K2+1)*S2
      B2=A(K2+1)*S2+B(K2+1)*C2
      A3=A(K3+1)*C3-B(K3+1)*S3
      B3=A(K3+1)*S3+B(K3+1)*C3
286  A(K0+1)=A0+A2+A1+A3
      B(K0+1)=B0+B2+B1+B3
      A(K1+1)=A0+A2-A1-A3
      B(K1+1)=B0+B2-B1-B3
      A(K2+1)=A0-A2-B1+B3
      B(K2+1)=B0-B2+A1-A3
      A(K3+1)=A0-A2+B1-B3
290  B(K3+1)=B0-B2-A1+A3
      IF (K .GT. 0) GO TO 296
      KB=K3+SPAN
      IF (KB .LT. KN) GO TO 298
C
C      LABEL 294 IS L6 IN ALGOL
C
294  IF (KN .LT. N) GO TO 250
      RETURN
296  K=K-2
      GO TO 275
298  IF (J .EQ. 0) GO TO 300

```

```

J=J-1
C2=C1
IF (J .EQ. 1) GO TO 302
C1=(C1-S1)*SQ
S1=(C2+S1)*SQ
GO TO 280
300 K=2
J=MK
GO TO 270
302 C1=C1*CK+S1*SK
S1=S1*CK-C2*SK
GO TO 280
END
SUBROUTINE REORDR (KS,REEL)
C
C      PROCEDURE REORDER (A,B,N,M,KS,REEL)
C      USES NO OTHER PROCEDURES
C
COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,B,T
INTEGER*4 M,N,KS,C(9),LST(9),NA,NAA
INTEGER*4 I,J,JJ,K,KB,K2,KU,LIM,P
LOGICAL INVRSE,REEL
C(M+1)=KS
NA=IABS(M-1)+1.0000001
DO 450 NAA=1,NA
K=M-NAA+1
450 C(K)=C(K+1)/2
J=M-1
P=J
KB=0
I=KB
IF (REEL) GO TO 454
M=M-1
GO TO 460
454 KU=N-2
NA=IABS(KU/2)+1.0000001
DO 458 NAA=1,NA
K=NAA*2-2
T=A(K+2)
A(K+2)=B(K+1)
458 B(K+1)=T
460 LIM=(M+2)/2
IF (P .LE. 0) RETURN
C
C      LABEL 464 IS L IN ALGOL
C
464 K2=C(J+1)+KB
KU=K2
JJ=C(M-J+1)
KK=KB+JJ
C
C      LABEL 468 IS L2 IN ALGOL
C
468 K=KK+JJ
C
C      LABEL 472 IS L3 IN ALGOL
C
472 T=A(KK+1)
A(KK+1)=A(K2+1)

```

```

A(K2+1)=T
T=B(KK+1)
B(KK+1)=B(K2+1)
B(K2+1)=T
KK=KK+1
K2=K2+1
IF (KK .LT. K) GO TO 472
KK=KK+JJ
K2=K2+JJ
IF (KK .LT. KU) GO TO 468
IF (J .LE. LIM) GO TO 476
J=J-1
I=I+1
LST(I+1)=J
GO TO 464
476 KB=K2
IF (I .LE. 0) GO TO 480
J=LST(I+1)
I=I-1
GO TO 464
480 IF (KB .GE. N) RETURN
J=P
GO TO 464
END

```

APPENDIX B

THE FORTRAN VERSION
OF BEAM FOR THE
NASA RADIATION PATTERN
PROGRAM

```

//
//C91540    JOB
//          (0643,EE,30,40),'BIM423',REGION=200K
// EXEC FORTHCLG,PARM.FORT='SOURCE,MAP,OPT=2'
//FORT.SYSIN DD *
C
C          *****DRIVER PROGRAM FOR COMPUTING RADIATION PATTERN*****
C
COMMON A,AXI,AXR,AYI,AYR,AZI,AZR,A1,A2,A3,A4,A5,A6,A7, B,AR,BR
COMMON CA,CB,CELNUM,CFI,CFR,CG,CKD,CKR,CRI,CRR,CTA,CTI,CELAST
COMMON COSTA,CPhi,SINTA,SPHI
COMMON D1,D2,D3,DIA,DPhi,DR,DRHO,DRR,DRI,DTR,DTI,DFR,DFI
COMMON E11,E12,E13,E21,E22,E23,E31,E32,E33,EX,EY,EZ
COMMON F,F1,FLDA,FNUM,FRING,FR,FRC,FZ,GR,G,BIG
COMMON H,HX,HY,HZ,HXF,HYF,HZF
COMMON IS,IE,IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IF1,IF2,IF3,IF4,IF5
COMMON IF6,IF7,IF8,N2,N3,N4,N5,N6,N7,IRM,L5,L6,L7,L8,L9,L10
COMMON I,I1,ID,IDSUM,IEND,IENDO,IFI,IN,IR,IR1,ISUM,ITI,IBIG
COMMON J,J1,JB1,JBIG,JRING,LIR,LIR1,LSW,L1,L2,L3,L4,M,I1ST
COMMON NFI,NFP,NPR,NR,NRING,NRING1,NRSG,NSUMDS,NT,NTA,NU,NLLR,NUM
COMMON PX,PY,PZ
COMMON PHI,POWER,Q,QD,QFI,QR,PHASE,R1,R2,RP,RHO,RHO2
COMMON SA,SB,SFI,SG,SKD,SKR,STA,TA,TPI,VDSJ
COMMON X,XI,XIUL,XR,XRUL,Y,YI,YIUL,YR,YRUL,Z,ZI,ZIUL,ZR,ZRUL
COMMON NRI(400),NSUM(400),NUMSUM(400),NUMT(250),NUMF(250)
COMMON CBSTD(250),CBSFD(250)
COMMON VX(1000),VY(1000),VZ(1000),VDS(1000),VHXR(1000),VHXI(1000)
COMMON VHXR(1000),VHYI(1000),VHZR(1000),VHZI(1000),VNXZ(1000)
COMMON VNYZ(1000)
DIMENSION PWR(250)
DIMENSION FV(250,6)
DIMENSION JCK(54)
C
C          *****BEGIN READING*****
C
29  READ(5,40)NTA,NFI,M,NU,L3
40  FORMAT(5I10)
    IF(NTA.EQ.0) GO TO 8060
    READ(5,41)DIA,DIA1,DIA2,A1,FRC,FLDA,F
41  FORMAT(4F10.5,F14.10,2F10.5)
    READ(5,403)A,B,G,EX,EY,EZ,D1,D2,D3
403  FORMAT(9F8.3)
    READ(5,402)(CBSTD(I),I=1,NTA)
    READ(5,402)(CBSFD(I),I=1,NFI)
402  FORMAT(8(1X,F8.3))
C
C          *****READ THE FIELD POINTS WHICH ARE NOT TO BE COMPLETED*****
C
877  READ(5,877)(JCK(J),J=1,54)
    FORMAT(18I4)
C
C          *****END OF READING*****
C
C          *****INITIALISE*****
C
DO 4321 I=1,250
DO 4321 J=1,6
FV(I,J)=0
4321. CONTINUE
DO 205 ID=1,250

```

```

205  PWR(ID)=0.0
      NUMSUM(1)=0
      L5=0
      L6=0
      JRING=1000
      LIST=1000

C
C      *****BEGIN PREFACE WRITING*****
C
      WRITE(6,1006)
1006  FORMAT(1H1)
      WRITE(6,9876)
9876  FORMAT(/3X,66HTHE DATA CARDS READ,THEIR CORRESPONDING PARAMETERS
1AND FORMAT ARE)
      WRITE(6,43)
43    FORMAT(/3X,72H1234567810123456782012345678301234567840123456785012
134567860123456787012)
      WRITE(6,44)
44    FORMAT(/3X,72HNO. TETAS NO. FIS    ARRAY SIZE CUS**N    DETAILS    YES
1OR NC
      WRITE(6,9874)NTA,NFI,M,NU,L3
9874  FORMAT(3X,5I10)
      WRITE(6,45)
45    FORMAT(/3X,76HDIAMETER    HOLE DIA1 HOLE DIA2 DEVIATION    SCALE
1WAVELENGTH    FOCAL DIST.)
      WRITE(6,46)DIA,DIA1,DIA2,A1,FRC,FLDA,F
46    FORMAT(3X,4F10.5,F14.10,2F10.5)
      WRITE(6,9871)
9871  FORMAT(3X,4HALFA,4X,4HBETA,4X,4HGAMA,7X,1HX,7X,1HY,7X,1HZ,7X,2HD1
1, 6X,2HD2,6X,2HD3)
      WRITE(6,9869)A,B,G,EX,EY,EZ,D1,D2,D3
9869  FORMAT(3X,9F8.3)
      WRITE(6,1003)
1003  FORMAT(1H )
      WRITE(6,9873)
9873  FORMAT(3X,28HTETA DEGREES OF FIELD POINTS)
      WRITE(6,4021)(CBSTD(I),I=1,NTA)
4021  FORMAT(3X,8(1X,F8.3))
      WRITE(6,1003)
      WRITE(6,9872)
9872  FORMAT(3X,26HF1 DEGREES OF FIELD POINTS)
      WRITE(6,4021)(CBSFD(I),I=1,NFI)
      WRITE(6,60)
60    FORMAT(/3X,56HFOLLOWING POINTS IN THE TETA-F1 MATRIX HAVE BEEN OMI
1TTED)
      WRITE(6,61)(JCK(J),J=1,54)
61    FORMAT(3X,18I4)

C
C      *****END OF PREFACE*****
C
C      *****CALCULATE ELEMENTS OF EULER MATRIX*****
C
      AR=A*DR
      CA=COS(AR)
      SA=SIN(AR)
      BR=B*DR
      CB=COS(BR)
      SB=SIN(BR)
      GR=G*DR
      CG=COS(GR)

```

```

SG=SIN(GR)
E11=CG*CA-CB*SA*SC
E21=-SG*CA-CB*SA*CG
E31=SB*SA
E12=CG*SA+CB*CA*SG
E22=-SG*SA+CB*CA*CG
E32=-SB*CA
E13=SG*SB
E23=CG*SB
E33=CB
PX=D1*E11+D2*E21+D3*E31
PY=D1*E12+D2*E22+D3*E32
PZ=D1*E13+D2*E23+D3*E33

C
C      *****BEGIN SEGMENTATION*****
C
D3=(1.-D1**2-D2**2)**.5
TPI=2.*3.141592654
DR=TPI/360
Q=TPI/FLDA
DRHO=FLDA/FRC
NR=(DIA/2.)/DRHO
L7=(DIA1/2.)/DRHO
L8=(DIA2/2.)/DRHO
WRITE(6,102)NR
102  FORMAT(          ///,3X,25HREFLECTOR IS DIVIDED INTO,14,7H RINGS.)
      I=0
      J=0
13    I=I+1
      IF(I.GT.999) GO TO 701
      ISUM=0
12    J=J+1
      IF(J.GT.NR) GO TO 10
      IDSUM=6*J
      IF(IDSUM.GT.M) GO TO 14
      ISUM=ISUM+IDSUM
      IF(ISUM.GT.M) GO TO 11
      GO TO 12
11    I1=I+1
      NRI(I1)=J-1
      NSUM(I1)=ISUM-IDSUM
      J=J-1
      GO TO 13
14    WRITE(6,103)J,M
103  FORMAT(/2X,37H THE NUMBER OF ELEMENTAL AREAS IN THE,14,20H RING 1
      IS LARGE THAN,15,42H .WILL CONSIDER PART OF RINGS AS SEGMENTS.)
      IF(ISUM.EQ.0) GO TO 876
      I1=I+1
      NSUM(I1)=ISUM
      NRI(I1)=J-1
      I=I1
      L6=1
876   L5=1
      JRING=J
      I1ST=I1+1
51    NDIV=IDSUM/M
      NREM=IDSUM-NDIV*M
      DO 511 ISK=1,NDIV
      I1=I1+1
      NRI(I1)=J

```

```

      NSUM(I1)=M
511  CONTINUE
      IF(NREM.EQ.0) GO TO 875
      I1=I1+1
      NRI(I1)=J
      NSUM(I1)=NREM
875  J=J+1
      IF(J.GT.NR) GO TO 10
      IDSUM=5*J
      GO TO 51
10   WRITE(6,104)
104  FORMAT(/2X,55H CONTRIBUTION OF ALL REFLECTOR RINGS WILL BE PROCES
1SED.)
      IF(L5.EQ.1) GO TO 515
      I1=I+1
      NRI(I1)=J-1
      NSUM(I1)=ISUM
515  I=I1
15   LIR=I
      LIRI=LIR-1
      NRI(1)=0
      NSUM(1)=0
      NSUMDS=0
      DO 201 IN=2,LIR
201  NSUMDS=NSUMDS+NSUM(IN)
      WRITE(6,1009) NSUMDS,LIRI,M
1009 FORMAT(/3X,35H THE TOTAL NO. OF AREAS IS NSUMDS = ,I8,
128H, NO. OF SEGMENTS IS LIRI = ,I3,6H, M = ,I4,2H .)
      WRITE(6,1008) DIA,FLDA,FRC
1008 FORMAT(/3X,37H RESULTS BASED ON INPUT DATA, DIA. = ,F8.4,
115H, WAVELENGTH = ,F8.4,31H ,SIDE OF ELEMENTAL AREA FRC = ,F8.5,
215H OF WAVELENGTH.)
      WRITE(6,1021) F,A,B,G,EX,EY,EZ
1021 FORMAT (/3X,2HF=,F8.3,6H,ALFA=,F8.3,6H,BETA=,F8.3,6H,GAMA=,F8.3,
120H,TRANSLATIONS ARE,X=,F8.3,3H,Y=,F8.3,3H,Z=,F8.3,2H .)
      WRITE(6,1031) D1,D2,D3
1031 FORMAT(/3X,34H THE POLARIZATION COSINES ARE D1 = ,F8.5,6H,D2 = ,
1F8.5,6H,D3 = ,F8.5,2H .)
      WRITE(6,7113)
7113 FORMAT(/3X,72H FOLLOWING ARE THE ORDER NUMBERS OF THE LAST RINGS I
IN SUCCESSIVE SEGMENTS.)
      WRITE(6,7114) (NRI(I),I=2,LIR)
7114 FORMAT(/ 21(3X,I3))
      WRITE(6,7115)
7115 FORMAT(/3X,67H FOLLOWING ARE THE NUMBER OF ELEMENTAL AREAS IN SUCC
1ESSIVE SEGMENTS.)
      WRITE(6,7114) (NSUM(I),I=2,LIR)

C
C      *****END OF SEGMENTATION*****
C
C      *****BEGIN PREPARATION FOR SETUP*****
C
C
      BIG=0.
      IR=0
      I4=0
16   IR=IR+1
C
C      *****ALL SEGMENTS DONE*****
C
      IF(IR.EQ.LIR) GO TO 300

```

```

WRITE(6,1003)
NFP=0
IR1=IR+1
IF(IR1.GE.I1ST) GO TO 5051
NLLR=NRI(IR)
NPR=NRI(IR1)
NRSG=NPR-NLLR
IEND=0
I=1
J=-1
1 J=J+1
IF(J.EQ.NRSG) GO TO 3
NRING=NRI(IR)+J+1
LSW=0
NRING1=NRING+1
FRING=FLOAT(NRING)
FRING=FRING-.5
RHU=FRING*DRHO
RHO2=RHO**2
I=I-1
IEND0=IEND
NUM=6*NRING
FNUM=NUM
DPHI=TPI/FNUM
IEND=IEND0+NUM
NUMSUM(NRING1)=IEND
IF(NRING.GT.L7.AND.NRING.LT.L8) GO TO 24
C
C *****SETUP RING BY RING*****
C
CALL SETUP
GO TO 1
24 I=IEND+1
GO TO 1
C
C *****RING CONTAINS MORE THAN ONE SEGMENT*****
C
5051 NPR=NRI(IR1)
NLLR=NPR-1
NRSG=1
NRING=NRI(IR1)
LSW=0
NRING1=NRING+1
FRING=FLOAT(NRING)
FRING=FRING-.5
RHU=FRING*DRHO
RHO2=RHO**2
NUM=6*NRING
IF(NRI(IR1).EQ.NRI(IR)) GO TO 53
CELNUM=-.5
CELAST=FLOAT(NSUM(IR1))
52 I=0
FNUM=NUM
DPHI=TPI/FNUM
NUMSUM(NRING1)=NSUM(IR1)
IEND=10000000
IF(NRING.GT.L7.AND.NRING.LT.L8) GO TO 25
C
C *****SETUP WHEN RING CONTAINS MOR THAN ONE SEGMENT*****
C

```

```

CALL SETUP
C
GO TO 3
25 I=IEND+1
GO TO 3
53 CELAST=CELAST+FLOAT(NSUM(IR1))
GO TO 52
C
C *****BEGIN WITH FIELD POINTS*****
C
3 JAK=0
DO 901 IFI=1,NFI
DO 901 ITI=1,NTA
JAK=JAK+1
DO 903 JA=1,54
903 IF(JAK.EQ.JCK(JA)) GO TO 901
NFP=NFP+1
NUMF(NFP)=IFI
NUMT(NFP)=ITI
C
C *****HEADING HAS BEEN WRITTEN*****
C
IF(L4.EQ.1) GO TO 2222
C
C *****PRINTING OF DETAILS NOT REQUIRED*****
C
IF(L3.EQ.1) GO TO 2222
C
C *****WRITE READING FOR DETAILED DATA TABLE*****
C
WRITE(6,1006)
WRITE(6,1003)
WRITE(6,1003)
WRITE(6,1034)
1034 FORMAT(3X,89H FOLLOWING TABLE GIVES VARIOUS FIELD VALUES FOR INDICA
TED FIELD POINTS AND SEGMENT NUMBERS)
WRITE(6,1208) DIA,FLDA,FRC
1208 FORMAT(/,3X,37H THEY ARE BASED ON INPUT DATA, DIA. = ,F8.4,
115H, WAVELENGTH = ,F8.4,31H ,SIDE OF ELEMENTAL AREA FRC = ,F8.5,
215H OF WAVELENGTH.)
WRITE(6,1021) F,A,B,G,EX,EY,EZ
WRITE(6,1031) D1,D2,D3
WRITE(6,1003)
WRITE(6,1003)
WRITE(6,5555)
5555 FORMAT(/,3X,12H FIELD VALUES,27X,3HERR,8X,3HERI,8X,3HETR,8X,3HETI,
18X,3HEFR,8X,3HEFI,6X,5HPOWER,4X,10HTETA PHASE)
WRITE(6,1003)
WRITE(6,5656)
5656 FORMAT(3X,34H POINT NO. TETA FI SEGMENT)
C
C *****END OF HEADER WRITING*****
C
C *****START INTEGRATION PROCEDURE*****
C
L4=1
2222 XR=0.
XI=0.
YR=0.

```

```

      YI=0.
      ZR=0.
      ZI=0.
      TA=CBSTD(ITI)*DR
      FI=CBSFD(IFI)*DR
      STA=SIN(TA)
      CTA=COS(TA)
      SFI=SIN(FI)
      CFI=COS(FI)

C
C      *****INTEGRATE*****
C
      CALL ADDUP
C
C      *****TRANSFORM TO SPHERICAL COORDINATES*****
C
      CRR=STA*CFI*XR+STA*SFI*YR+CTA*ZR
      CRI=STA*CFI*XI+STA*SFI*YI+CTA*ZI
      CFR=CFI*YR-SFI*XR
      CFI=CFI*YI-SFI*XI
      CTR=CTA*CFI*YR+CTA*SFI*YR-STA*ZR
      CTI=CTA*CFI*XI+CTA*SFI*YI-STA*ZI
      FV(NFP,1)=FV(NFP,1)+CRR
      FV(NFP,2)=FV(NFP,2)+CRI
      FV(NFP,3)=FV(NFP,3)+CTR
      FV(NFP,4)=FV(NFP,4)+CTI
      FV(NFP,5)=FV(NFP,5)+CFR
      FV(NFP,6)=FV(NFP,6)+CFI
      IF(FV(NFP,3).EQ.0.0 .OR. FV(NFP,4).EQ. 0.0) GO TO 27
      PHASE=ATAN2(FV(NFP,3),FV(NFP,4))/DR
      GO TO 28
27  PHASE=0.0
28  POWER      =FV(NFP,3)**2+FV(NFP,4)**2+FV(NFP,5)**2+FV(NFP,6)**2
      IF(IR1.NE.LIR) GO TO 55
      PWR(NFP)  =FV(NFP,3)**2+FV(NFP,4)**2+FV(NFP,5)**2+FV(NFP,6)**2

C
C      *****DETAILS OF DATA NOT REQUIRED*****
C
55  IF(L3.EQ.1) GO TO 901
C
C      *****WRITE COMPONENTS OF ELECTRIC FIELD*****
C
      WRITE(6,5655)NFP,CBSTD(ITI),CBSFD(IFI),IR,FV(NFP,1),FV(NFP,2),
1  FV(NFP,3),FV(NFP,4),FV(NFP,5),FV(NFP,6),POWER      ,PHASE
5655  FORMAT(3X,I3,5X,F7.2,1X,F7.2,3X,I3,3X,8(F10.2,1X))
901  CONTINUE
C
C      *****START WITH A NEW SEGMENT*****
C
      GO TO 16
C
C      *****ALL SEGMENTS AND FIELD POINTS DONE*****
C
C      *****FIND THE DIRECTION OF MAXIMUM RADIATED POWER*****
C
300  DO 500 I=1,NFP
      IF(PWR(I).GT.BIG) GO TO 501
      GO TO 500
501  IBIG=I
      BIG=PWR(I)

```

```

500  CONTINUE
      DO 502 I=1,NFP
      IF(PWR(I).EQ.0.0) PWR(I)=0.000000001
      PWR(I)=10.*ALOG10(PWR(I)/BIG)
502  CONTINUE
      IFI=NUMF(IBIG)
      ITI=NUMT(IBIG)

C
C      *****END OF COMPUTATION*****
C
C      *****WRITE HEADING FOR DB TABLE*****
C
      IS=1
      IE=8
      NTAB=(NFP-1)/8+1
      WRITE(6,1006)
      WRITE(6,1010)CBSTD(ITI),CBSFD(IFI)
1010  FORMAT(/3X,46HMAXIMUM POWER IS RADIATED IN DIRECTION TETA = ,F8.3
1,5H,FI= ,F8.3)
      WRITE(6,1008)DIA,FLDA,FRC
      WRITE(6,1021)F,A,B,G,EX,EY,EZ
      WRITE(6,1031)D1,D2,D3
      WRITE(6,3333)
3333  FORMAT(/3X,118HIN THE FOLLOWING TABLE EACH ROW GIVES THE POWER IN
1 DB. THE ZERO DB REFERENCE IS THE POWER RADIATED IN THE DIRECTIO
2N )
      WRITE(6,3334)CBSTD(ITI),CBSFD(IFI),BIG
3334  FORMAT(/3X,7HTETA = ,F8.3,10H AND FI = ,F8.3,25H AND HAS ABSOLUTE
1 VALUE ,F12.3)
      DO 208 I=1,NTAB
      N2=IS+1
      N3=N2+1
      N4=N3+1
      N5=N4+1
      N6=N5+1
      N7=N6+1
      WRITE(6,6666)IS,N2,N3,N4,N5,N6,N7,IE
6666  FORMAT(/3X,18HFIELD POINT ,2X,8(I3,9X))
      IT1=NUMT(IS)
      IT2=NUMT(N2)
      IT3=NUMT(N3)
      IT4=NUMT(N4)
      IT5=NUMT(N5)
      IT6=NUMT(N6)
      IT7=NUMT(N7)
      IT8=NUMT(IE)
      IF1=NUMF(IS)
      IF2=NUMF(N2)
      IF3=NUMF(N3)
      IF4=NUMF(N4)
      IF5=NUMF(N5)
      IF6=NUMF(N6)
      IF7=NUMF(N7)
      IF8=NUMF(IE)
      WRITE(6,9222)CBSTD(IT1),CBSTD(IT2),CBSTD(IT3),CBSTD(IT4),CBSTD(IT5
1),CBSTD(IT6),CBSTD(IT7),CBSTD(IT8)
9222  FORMAT(3X,12HTETA DEGREES,6X,8F12.6)
      WRITE(6,9333)CBSFD(IF1),CBSFD(IF2),CBSFD(IF3),CBSFD(IF4),CBSFD(IF5
1),CBSFD(IF6),CBSFD(IF7),CBSFD(IF8)
9333  FORMAT( 3X,10HFI DEGREES,8X,8F12.6)

```

```

WRITE(6,1003)
WRITE(6,1003)
9672 WRITE(6,1001)(PWR(J),J=IS,IE)
1001 FORMAT(3X,11HDB LEVEL ,4X,3X,8F12.6)
9673 IS=IS+8
      IE=IE+8
208  CONTINUE
      WRITE(15,8000)
8000  FORMAT(5X,3HPHI,10X,5HTHETA,10X,2HDB)
      WRITE(15,8002) NFP
8002  FORMAT(3X,I3)
      DO 8050 I=1,NFP
      KT=NUMT(I)
      KF=NUMF(I)
8050  WRITE(15,8001) CBSFD(KF),CBSTD(KT),PWR(I)
8001  FORMAT(4X,F12.6,3X,F12.6,3X,F12.6)
      GO TO 29
8060  WRITE(15,8061)
8061  FORMAT(5X,3HEND)
      RETURN
701  WRITE(6,7111)
7111  FORMAT(/,3X,35HTHE RING DIMENSION IS INSUFFICIENT.)
      RETURN
      END
      SUBROUTINE SETUP
      COMMON A,AXI,AXR,AYI,AYR,AZI,AZR,A1,A2,A3,A4,A5,A6,A7, B,AR,BR
      COMMON CA,CB,CELCUM,CFI,CFR,CG,CKD,CKR,CRI,CRR,CTA,CTI,CELAST
      COMMON COSTA,CPhi,SINTA,SPHI
      COMMON D1,D2,D3,DIA,DPhi,DR,DRHO,DRR,DRI,DTR,DTI,DFR,DFI
      COMMON E11,E12,E13,E21,E22,E23,E31,E32,E33,EX,EY,EZ
      COMMON F,F1,FLDA,FNUM,FRING,FR,FRC,FZ,GR,G,BIG
      COMMON H,HX,HY,HZ,HXF,HYF,HZF
      COMMON IS,IE,IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IF1,IF2,IF3,IF4,IF5
      COMMON IF6,IF7,IF8,N2,N3,N4,N5,N6,N7,IRM,L5,L6,L7,L8,L9,L10
      COMMON I,I1,ID,IDSUM,IEND,IENDO,IFI,IN,IR,IR1,ISUM,ITI,IBIG
      COMMON J,J1,JB1,JBIG,JRING,LIR,LIR1,LSW,L1,L2,L3,L4,M,I1ST
      COMMON NFI,NFP,NPR,NR,NRING,NRING1,NRSG,NSUMDS,NT,NTA,NU,NLLR,NUM
      COMMON PX,PY,PZ
      COMMON PHI,POWER,Q,QD,QFI,QR,PHASE,R1,R2,RP,RHO,RHO2
      COMMON SA,SB,SFI,SG,SKD,SKR,STA,TA,TPI,VDSJ
      COMMON X,XI,XIUL,XR,XRUL,Y,YI,YIUL,YR,YRUL,Z,ZI,ZIUL,ZR,ZRUL
      COMMON NRI(400),NSUM(400),NUMSUM(400),NUMT(250),NUMF(250)
      COMMON CBSTD(250),CBSFD(250)
      COMMON VX(1000),VY(1000),VZ(1000),VDS(1000),VHXR(1000),VHXI(1000)
      COMMON VHXR(1000),VHYI(1000),VHZR(1000),VHZI(1000),VNXZ(1000)
      COMMON VNYZ(1000)
2     I=I+1
      IF(I.EQ.(IEND+1)) RETURN
      IF(IR1.GE.I1ST) GO TO 17
      CELCUM=FLOAT(I-IENDO)
      CELCUM=CELCUM-.5
18    PHI=DPhi*CELCUM
      CPhi=COS(PHI)
      SPhi=SIN(PHI)
      X=RHO*CPhi
      XEX=X-EX
      VX(I)=X
      Y=RHO*SPhi
      YEY=Y-EY
      VY(I)=Y

```

```

IF(LSW.EQ.1) GO TO 31
VZ(NRING)=RHO2/(F*4.)-F
Z=VZ(NRING)
ZEZ=Z-EZ
R2=RHO2+Z**2
R1=R2**.5
ZR1=Z-R1
VDS(NRING)=TPI*((RHO+.5*DRHO)**2-(RHO-.5*DRHO)**2)/(FNUM*2.)
LSW=1
31 VNXZ(I)=X/ZR1
VNYZ(I)=Y/ZR1
C
C *****RP IS THE DISTANCE FROM THE PHASE CENTER TO ELEMENTAL A
C
RP=(XEX**2+YEY**2+ZYZ**2)**.5
COSTA=(E31*XEX+E32*YEY+E33*ZEZ)/RP
C
C *****FR IS = COS TETA**NU/RP, THE PATTERN FACTOR OF SOURCE**
C
FR=(COSTA**NU)/RP
CR=Q*RP-A1*DR*(1.-COSTA)
CKR=COS(CR)
SKR=SIN(CR)
C
C *****HX, HY, HZ ARE THE COMPONENTS OF H IN DIRECTION OF H FIEL
C
HX=YEY*PZ-ZEZ*PY
HY=ZYZ*PX-XEX*PZ
HZ=XEX*PY-YEY*PX
H=(HX**2+HY**2+HZ**2)**.5
HXF=HX*FR/H
HYF=HY*FR/H
HZF=HZ*FR/H
22 VHXR(I)= HXF*CKR
VHXI(I)=-HXF*SKR
VHYR(I)= HYF*CKR
VHYI(I)=-HYF*SKR
VHZR(I)= HZF*CKR
VHZI(I)=-HZF*SKR
GO TO 2
17 CELNUM=CELCUM+1.
IF(CELCUM.GT.CELAST) GO TO 19
GO TO 18
19 CELNUM=CELCUM-1.
RETURN
END
SUBROUTINE ADDUP
COMMON A,AXI,AXR,AYI,AYR,AZI,AZR,A1,A2,A3,A4,A5,A6,A7, B,AR,BR
COMMON CA,CB,CELCUM,CFI,CFR,CG,CKD,CKR,CRI,CRR,CTA,CTI,CELCST
COMMON COSTA,CPHI,SINTA,SPHI
COMMON D1,D2,D3,DIA,DPHI,DR,DRHO,DRR,DRI,DTR,DTI,DFR,DFI
COMMON E11,E12,E13,E21,E22,E23,E31,E32,E33,EX,EY,EZ
COMMON F,F1,FLDA,FNUM,FRING,FR,FRC,FZ,GR,G,BIG
COMMON H,HX,HY,HZ,HXF,HYF,HZF
COMMON IS,IE,IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IF1,IF2,IF3,IF4,IF5
COMMON IF6,IF7,IF8,N2,N3,N4,N5,N6,N7,IRM,L5,L6,L7,L8,L9,L10
COMMON I,I1,ID,IDSUM,IEND,IENDU,IFI,IN,IR,IR1,ISUM,ITI,IBIG
COMMON J,J1,JB1,JBIG,JRING,LIR,LIR1,LSW,L1,L2,L3,L4,M,I1ST
COMMON NFI,NFP,NPR,NR,NRING,NRING1,NRSG,NSUMDS,NT,NTA,NU,NLLR,NUM
COMMON PX,PY,PZ

```

```

COMMON PHI,POWER,Q,QD,QFI,QR,PHASE,R1,R2,RP,RHO,RHO2
COMMON SA,SB,SFI,SG,SKD,SKR,STA,TA,TPI,VDSJ
COMMON X,XI,XIUL,XR,XRUL,Y,YI,YIUL,YR,YRUL,Z,ZI,ZIUL,ZR,ZRUL
COMMON NRI(400),NSUM(400),NUMSUM(400),NUMT(250),NUMF(250)
COMMON CBSTD(250),CBSFD(250)
COMMON VX(1000),VY(1000),VZ(1000),VDS(1000),VHXR(1000),VHXI(1000)
COMMON VHYZ(1000),VHYI(1000),VHZR(1000),VHZI(1000),VNXZ(1000)
COMMON VNYZ(1000)
I=0
J=NLLR
7 J=J+1
J1=J+1
IF(J.GT.NPR) RETURN
VDSJ=VDS(J)
XRUL=0.
XIUL=0.
YRUL=0.
YIUL=0
ZRUL=0.
ZIUL=0.
37 I=I+1
IF(I.GT.NUMSUM(J1)) GO TO 38
IF(J.GT.L7.AND.J.LT.L8) GO TO 37
CD=Q*(VX(I)*CFI*STA+VY(I)*SFI*STA+VZ(J)*CTA)
CKD=CDS(CD)
SKD=SIN(CD)
AXR=VNYZ(I)*VHZR(I)-VHYR(I)
AXI=VNYZ(I)*VHZI(I)-VHYI(I)
XRUL=XRUL+(AXR*CKD-AXI*SKD)
XIUL=XIUL+(AXR*SKD+AXI*CKD)
AYR=VHXR(I)-VNXZ(I)*VHZR(I)
AYI=VHXI(I)-VNXZ(I)*VHZI(I)
YRUL=YRUL+(AYR*CKD-AYI*SKD)
YIUL=YIUL+(AYR*SKD+AYI*CKD)
AZR=VNXZ(I)*VHYR(I)-VNYZ(I)*VHXR(I)
AZI=VNXZ(I)*VHYI(I)-VNYZ(I)*VHXI(I)
ZRUL=ZRUL+(AZR*CKD-AZI*SKD)
ZIUL=ZIUL+(AZR*SKD+AZI*CKD)
GO TO 37
38 XR=XR+XRUL*VDSJ
XI=XI+XIUL*VDSJ
YR=YR+YRUL*VDSJ
YI=YI+YIUL*VDSJ
ZR=ZR+ZRUL*VDSJ
ZI=ZI+ZIUL*VDSJ
I=I-1
GO TO 7
END

/*
//GO.FT07F001 DD SYSOUT=B,DCB=(RECFM=F,BLKSIZE=80)
//GO.FT06F001 DD SYSOUT=A,DCB=(RECFM=UA,BLKSIZE=133)
//GO.FT15F001 DD SYSOUT=A,DCB=(RECFM=UA,BLKSIZE=133)
//GO.FT05F001 DD *

```

	16	1	999	1	0		
30.00000	0.00000	0.00000	0.00000	0.00000	4.731666667	0.42500	36.00000
0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000
0.000	1.000	2.000	3.000	4.000	5.000	6.000	7.000
8.000	9.000	10.000	11.000	12.000	13.000	14.000	15.000
0.000							